CHAPTER 4

Inference: Bucket Elimination for Probabilistic Networks

Having investigated bucket elimination in deterministic constraint networks in the previous chapter, we now present the bucket-elimination algorithm for the three primary queries defined over probabilistic networks: (1) belief-updating or computing posterior marginals (bel) as well as finding the probability of evidence; (2) finding the most probable explanation (mpe); and (3) finding the maximum a posteriori hypothesis (map).

We start focusing on queries over Bayesian networks first, and later show that the algorithms we derive are applicable with minor changes to Markov networks, cost networks, and mixed networks as well to the general graphical model. Below we recall the definition of Bayesian networks for your convenience.

Definition 4.1 **(Bayesian networks)** A *Bayesian network (BN)* is a 4-tuple $\mathcal{B} = \langle \mathbf{X}, \mathbf{D}, \mathbf{P}_G, \prod \rangle$. $\mathbf{X} = \{X_1, \ldots, X_n\}$ is a set of ordered variables defined over domains $\mathbf{D} = \{D_1, \ldots, D_n\}$, where $d = (X_1, \ldots, X_n)$ is an ordering of the variables. The set of functions $P_G = \{P_1, \ldots, P_n\}$ consist of conditional probability tables (CPTs for short) $P_i = \{P(X_i | \mathbf{Y}_i)\}$ where $\mathbf{Y}_i \subseteq \{X_{i+1}, ..., X_n\}$. These P_i functions can be associated with a directed acyclic graph G in which each node represents a variable X_i and $\mathbf{Y}_i = pa(X_i)$ are the parents of X_i in G. That is, there is a directed arc from each parent variable of X_i to X_i. The Bayesian network $\mathcal{B}$ represents the probability distribution over $\mathbf{X}$, $P_{\mathcal{B}}(x_1, \ldots, x_n) = \prod_{i=1}^{n} P(x_i | \mathbf{x}_{pa(X_i)})$. We define an evidence set e as an instantiated subset of the variables.

4.1 BELIEF UPDATING AND PROBABILITY OF EVIDENCE

Belief updating is the primary inference task over Bayesian networks. The task is to determine the posterior probability of singleton variables once new evidence arrives. For instance, if we are interested in the likelihood that the sprinkler was on last night (as we were in the Bayesian network example in Chapter 2), then we need to update this likelihood if we observe that the pavement near the sprinkler is slippery. More generally, we are sometime asked to compute the posterior marginals of a subset of variables given some evidence. Another important query over Bayesian networks, computing the probability of the evidence, namely computing the joint likelihood of a specific assignment to a subset of variables, is highly related to belief updating. We will show in this chapter how these tasks can be computed by the bucket-elimination scheme.

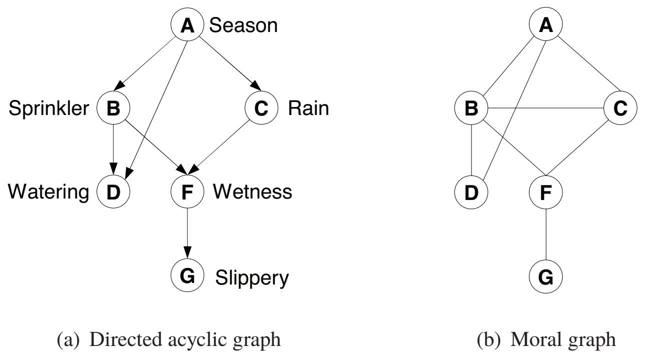


Figure 4.1: Belief network $P(G, F, C, B, A) = P(G|F)P(F|C, B)P(D|A, B)P(C|A)P(B|A)P(A)$.

4.1.1 DERIVING BE-BEL

We next present a step-by-step derivation of a general variable-elimination algorithm for belief updating. This algorithm is similar to adaptive-consistency, but the join and project operators of adaptive-consistency are replaced, respectively, with the operations of product and summation. We begin with an example and then proceed to describe the general case.

Let $X_1 = x_1$ be an atomic proposition (e.g., pavement = slippery). The problem of belief updating is to compute the conditional probability of X_1 given evidence $\mathbf{e}$, $P(x_1|\mathbf{e})$, and the probability of the evidence $P(\mathbf{e})$. By Bayes rule we have that $P(x_1|\mathbf{e}) = \frac{P(x_1,\mathbf{e})}{P(\mathbf{e})}$, where $\frac{1}{P(\mathbf{e})}$ is called the normalization constant. To develop the algorithm, we will use the previous example of a Bayesian network, (Figure 2.5), and assume the evidence is $G = 1$ (we also use $g = 1$ meaning that the specific assignment g is the number 1). For convenience we depict here part of the network again in Figure 4.1.

Consider the variables in the ordering $d_1 = A, C, B, F, D, G$. We want to compute $P(A = a|g = 1)$ or $P(A = a, g = 1)$. By definition,

$$P(a, g = 1) = \sum_{c,b,e,d,g=1} P(a, b, c, d, e, g) = \sum_{c,b,f,d,g=1} P(g|f)P(f|b, c)P(d|a, b)P(c|a)P(b|a)P(a).$$

We can now apply some simple symbolic manipulation, migrating each conditional probability table to the left of the summation variables that it does not reference. We get

$$P(a, g = 1) = P(a)\sum_{c} P(c|a)\sum_{b} P(b|a)\sum_{f} P(f|b, c)\sum_{d} P(d|b, a)\sum_{g=1} P(g|f). \qquad (4.1)$$

Carrying the computation from right to left (from G to A), we first compute the rightmost summation, which generates a function over F that we denote by $\lambda_G(F)$, defined by: $\lambda_G(f) = \sum_{g=1} P(g|f)$ and place it as far to the left as possible, yielding

$$P(a, g = 1) = P(a) \sum_c P(c|A) \sum_b P(b|a) \sum_f P(f|b, c)\lambda_G(f) \sum_d P(d|b, a). \tag{4.2}$$

(We index a generated function by the variable that was summed over to create it; for example, we created $\lambda_G(f)$ by summing over G.) Summation removes or eliminates a variable from the calculation.

Summing next over D (generating a function denoted $\lambda_D(B, A)$, defined by $\lambda_D(a, b) = \sum_d P(d|a, b)$), we get

$$P(a, g = 1) = P(a) \sum_c P(c|a) \sum_b P(b|a)\lambda_D(a, b) \sum_f P(f|b, c)\lambda_G(f) . \tag{4.3}$$

Next, summing over F (generating $\lambda_F(B, C)$ defined by $\lambda_F(b, c) = \sum_f P(f|b, c)\lambda_G(f)$), we get,

$$P(a, g = 1) = P(a) \sum_c P(c|a) \sum_b P(b|a)\lambda_D(a, b)\lambda_F(b, c) . \tag{4.4}$$

Summing over B (generating $\lambda_B(A, C)$), we get

$$P(a, g = 1) = P(a) \sum_c P(c|a)\lambda_B(a, c) . \tag{4.5}$$

Finally, summing over C (generating $\lambda_C(A)$), we get

$$P(a, g = 1) = P(a)\lambda_C(a) . \tag{4.6}$$

Summing over the values of variable A, we generate $P(g = 1) = \sum_a P(a)\lambda_C(a)$. The answer to the query $P(a|g = 1)$ can be computed by normalizing the last product in Eq. (4.6). Namely, $P(a|g = 1) = \alpha P(a)\lambda_C(a)$ where $\alpha = \frac{1}{P(g=1)}$.

We can create a bucket-elimination algorithm for this calculation by mimicking the above algebraic manipulation, using *buckets* as the organizational device for the various sums. First, we partition the $CPTs$ into buckets relative to the given order, $d_1 = A, C, B, F, D, G$. In bucket G we place all functions mentioning G. From the remaining $CPTs$ we place all those mentioning D in $bucket_D$, and so on. This is precisely the partition rule we used in the adaptive-consistency algorithm for constraint networks. This results in the initial partitioning given in Figure 4.2. Note that observed variables are also placed in their corresponding bucket.

Initializing the buckets corresponds to deriving the expression in Eq. (4.1). Now we process the buckets from last to first (or top to bottom in the figures), implementing the right to left computation in Eq. (4.1). Processing a bucket amounts to eliminating the variable in the bucket from subsequent computation. The $bucket_G$ is processed first. We eliminate G by summing over all values of G, but since we have observed that $G = 1$, the summation is over a singleton value. The function $\lambda_G(f) = \sum_{g=1} P(g|f) = P(g = 1|f)$, is computed and placed in $bucket_F$. In our calculations above, this corresponds to deriving Eq. (4.2) from Eq. (4.1)). Once we have created a new

$$\begin{aligned} Bucket_G = &\ P(g|f), g = 1 \\ Bucket_D = &\ P(d|b,a) \\ Bucket_F = &\ P(f|b,c) \\ Bucket_B = &\ P(b|a) \\ Bucket_C = &\ P(c|a) \\ Bucket_A = &\ P(a) \end{aligned}$$

Figure 4.2: Initial partitioning into buckets using $d_1 = A, C, B, F, D, G$.

function, it is placed in a lower bucket in accordance with the same rule we used to partition the original $CPTs$.

Following order d_1, we proceed by processing $bucket_D$, summing over D the product of all the functions that are in its bucket. Since there is a single function, the resulting function is $\lambda_D(b,a) = \sum_d P(d|b,a)$ and it is placed in $bucket_B$. Subsequently, we process the buckets for variables F, B, and C in order, each time summing over the relevant variable and moving the generated function into a lower bucket according to the same placement rule. In $bucket_A$ we compute the answer $P(a|g=1) = \alpha \cdot P(a) \cdot \lambda_C(a)$. Figure 4.3 summarizes the flow of functions generated during the computation.

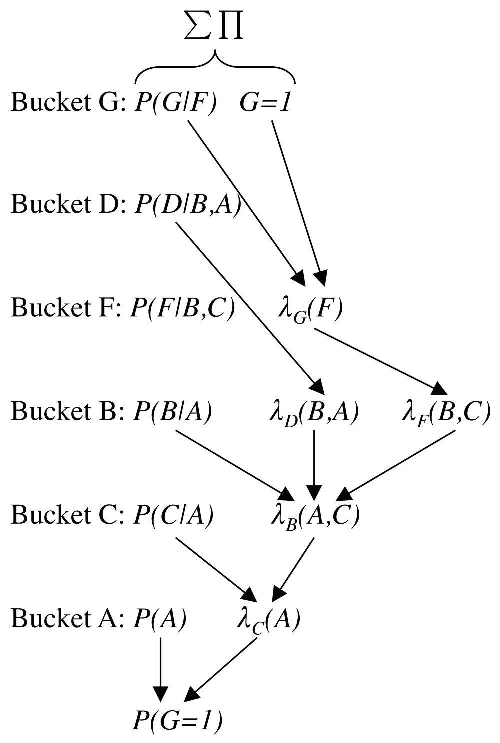
In this example, the generated λ functions were at most two-dimensional; thus, the complexity of processing each bucket using ordering d_1 is (roughly) time and space quadratic in the domain sizes. But would this also be the case had we used a different variable ordering? Consider ordering $d_2 = A, F, D, C, B, G$. To enforce this ordering we require that the summations remain in order d_2 from right to left, yielding (and we leave it to you to figure how the λ functions are generated):

$P(a, g=1) = P(a) \sum_f \sum_d \sum_c P(c|a) \sum_b P(b|a)\ P(d|a,b) P(f|b,c) \sum_{g=1} P(g|f)$
$= P(a) \sum_f \lambda_G(f) \sum_d \sum_c P(c|a) \sum_b P(b|a)\ P(d|a,b) P(f|b,c)$
$= P(a) \sum_f \lambda_G(f) \sum_d \sum_c P(c|a) \lambda_B(a,d,c,f)$
$= P(a) \sum_f \lambda_g(f) \sum_d \lambda_C(a,d,f)$
$= P(a) \sum_f \lambda_G(f) \lambda_D(a,f)$
$= P(a) \lambda_F(a)$.

The analogous bucket-elimination schematic process for this ordering is shown in Figure 4.4a. As before, we finish by calculating $P(a|g=1) = \alpha P(a) \lambda_F(a)$, where $\alpha = \frac{1}{\sum_a P(a)\lambda_F(a)}$.

We conclude this section with a general derivation of the bucket-elimination algorithm for probabilistic networks, called *BE-bel*. As a byproduct, the algorithm yields the probability of the evidence. Consider an ordering of the variables $d = (X_1, ..., X_n)$ and assume we seek $P(X_1|e)$. Note that if we seek the belief for variable X_1 it should initiate the ordering. Later we will see how this requirement can be relaxed. We need the definition of ordered-restricted Markov blanket.

Using the notation $\mathbf{x}_{(i..j)} = (x_i, x_{i+1}, ..., x_j)$ and $\mathbf{x} = (x_1, ..., x_n)$ we want to compute:



∑∏
Bucket G: P(G|F) G=1
Bucket D: P(D|B,A)
Bucket F: P(F|B,C) λ_G(F)
Bucket B: P(B|A) λ_D(B,A) λ_F(B,C)
Bucket C: P(C|A) λ_B(A,C)
Bucket A: P(A) λ_C(A)
P(G=1)

Figure 4.3: Bucket elimination along ordering $d_1 = A, C, B, F, D, G$.

$$P(x_1, \mathbf{e}) = \sum_{\mathbf{x}_{(2..n)}} P(\mathbf{x}, \mathbf{e}) = \sum_{\mathbf{x}_{(2..n-1)}} \sum_{x_n} \prod_i P(x_i, \mathbf{e}|\mathbf{x}_{pa_i}) \, .$$

Separating X_n from the rest of the variables, and denoting by F_j the parents of X_j including X_j, results in:

$$= \sum_{\mathbf{x}_{(2..n-1)}} \prod_{X_i \in X - F_n} P(x_i, \mathbf{e}|\mathbf{x}_{pa_i}) \cdot \sum_{x_n} \prod_{X_i \in F_n} P(x_i, \mathbf{e}|\mathbf{x}_{pa_i})$$

$$= \sum_{\mathbf{x}_{(2..n-1)}} \prod_{X_i \in X - F_n} P(x_i, \mathbf{e}|\mathbf{x}_{pa_i}) \cdot \lambda_n(\mathbf{x}_{S_n})$$

where

$$\lambda_n(\mathbf{x}_{S_n}) = \sum_{x_n} \prod_{X_i \in F_n} P(x_i, \mathbf{e}|\mathbf{x}_{pa_i}). \qquad (4.7)$$

The process continues recursively with X_{n-1}.

Thus, the computation performed in bucket X_n is captured by Eq. (4.7). Given ordering $d = X_1, ..., X_n$, where the queried variable appears first, the CPTs are partitioned using the rule

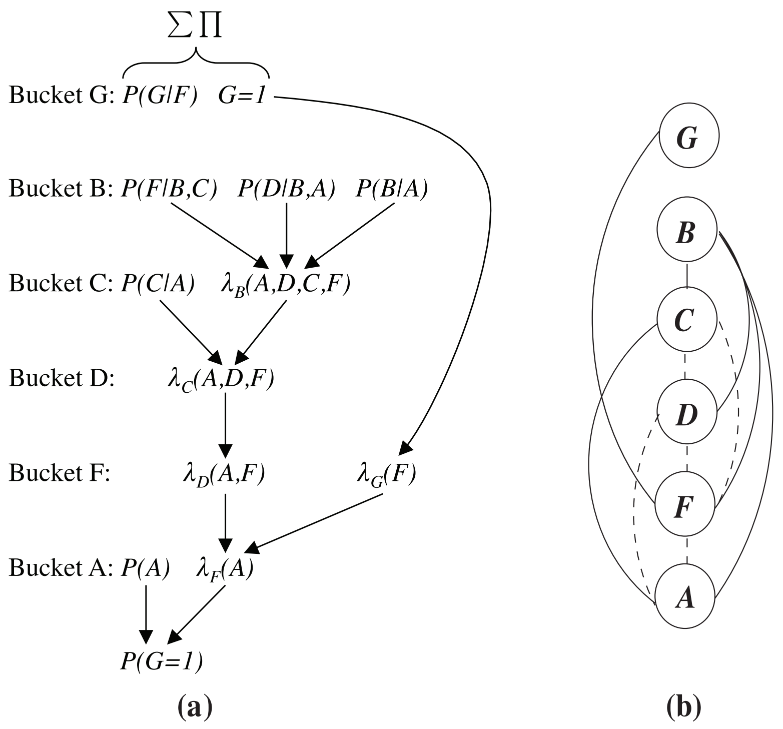


Figure 4.4: The bucket's output when processing along $d_2 = A, F, D, C, B, G$.

described earlier. The computed function in $bucket_{X_p}$ is λ_{X_p} and is defined over the bucket's scope, excluding X_p $\mathbf{S}_p = \cup_{\lambda_i \in bucket_{X_P}} scope(\lambda_i) - X_p$ by $\lambda_X = \sum_{X_p} \psi_X \cdot \prod_{\lambda \in bucket_{X_p}} \lambda$, where, $\psi_X = \prod_{P \in bucket_X} P$. This function is placed in the bucket of its largest-index variable in $\mathbf{S}_p$. Once processing reaches the first bucket, we have all the information to compute the answer which is the product of those functions. If we also *process* the first bucket we get the probability of the evidence. Algorithm BE-bel is described in Figure 4.5 (step 3 will elaborate more shortly). With the above derivation we showed the following.

Theorem 4.2 Sound and complete. *Algorithm BE-Bel applied along any ordering that starts with X_1 computes the belief $P(X_1|\mathbf{e})$. It also computes the probability of evidence $P(\mathbf{e})$ as the inverse of the normalizing constant in the first bucket.* $\square$

The bucket's operations for BE-bel

Processing a bucket requires the two types of operations on the functions in the buckets, combinations, and marginalization. The combination operation in this case is a product, which generates a

```
ALGORITHM BE-BEL
Input: A belief network $\mathcal{B} = \langle \mathbf{X}, \mathbf{D}, \mathbf{P}_G, \prod \rangle$, an ordering $d = (X_1, \ldots, X_n)$ ; evidence $e$
output: The belief $P(X_1|\mathbf{e})$ and probability of evidence $P(\mathbf{e})$
1.  Partition the input functions (CPTs) into $bucket_1$, ..., $bucket_n$ as follows: for
    $i \leftarrow n$ downto 1, put in $bucket_i$ all unplaced functions mentioning $X_i$. Put each
    observed variable in its bucket. Denote by $\psi_i$ the product of input functions in
    $bucket_i$.
2.  backward: for $p \leftarrow n$ downto 1 do
3.  for all the functions $\psi_{S_0}, \lambda_{S_1}, \ldots, \lambda_{S_j}$ in $bucket_p$ do
        If (observed variable) $X_p = x_p$ appears in $bucket_p$,
        assign $X_p = x_p$ to each function in $bucket_p$ and then
        put each resulting function in the bucket of the closest variable in its scope.
        else,
4.          $\lambda_p \leftarrow \sum_{X_p} \psi_p \cdot \prod_{i=1}^{j} \lambda_{S_i}$
5.          place $\lambda_p$ in bucket of the latest variable in scope($\lambda_p$),
6.  return (as a result of processing $bucket_1$):
        $P(\mathbf{e}) = \alpha = \sum_{X_1} \psi_1 \cdot \prod_{\lambda \in bucket_1} \lambda$
        $P(X_1|\mathbf{e}) = \frac{1}{\alpha} \psi_1 \cdot \prod_{\lambda \in bucket_1} \lambda$
```

Figure 4.5: BE-bel: a sum-product bucket-elimination algorithm.

function whose scope is the union of the scopes of the bucket's functions. The marginalization operation is summation, summing out the bucket's variable. The algorithm often referred to as being a *sum-product algorithm*.

Example 4.3 Let's look at an example of both of these operations in a potential bucket of B assuming it contains only two functions, $P(F|B,C)$ and $P(B|A)$. These functions are displayed in Figure 4.6. To take the product of the functions $P(F|B,C)$ and $P(B|A)$ we create a function over F, B, C, A where for each tuple assignment, the function value is the product of the respective entries in the input functions. To eliminate variable B by summation, we sum the function generated by the product, over all values in of variable B. We say that we *sum out* variable B. The computation of both the product and summation operators are depicted in Figure 4.7.

B	C	F	$P(F\|B,C)$
false	false	true	0.1
true	false	true	0.9
false	true	true	0.8
true	true	true	0.95

A	B	$P(B\|A)$
summer	false	0.2
fall	false	0.6
winter	false	0.9
spring	false	0.4

Figure 4.6: Examples of functions in the bucket of B.

The implementation details of the algorithm to perform these operations might have a significant impact on the performance. In particular, much depends on how the bucket's functions are represented. If, for example, the $CPTs$ are represented as matrices, then we can exploit efficient matrix multiplication algorithms. This important issue is outside the scope of this book.

4.1.2 COMPLEXITY OF BE-bel

Although BE-bel can be applied along any ordering, its complexity varies considerably across different orderings. Using ordering d_1 we recorded λ functions on pairs of variables only, while using d_2 we had to record functions on as many as four variables (see $Bucket_C$ in Figure 4.4a). The arity (i.e., the scope size) of the function generated during processing of a bucket equals the number of variables appearing in that processed bucket, excluding the bucket's variable itself. Since computing and recording a function of arity r is time and space exponential in r we conclude that the complexity of the algorithm is dominated by its largest scope bucket and it is therefore exponential in the size (number of variables) of the bucket having the largest number of variables. The base of the exponent is bounded by a variable's domain size.

Fortunately, as was observed earlier for adaptive-consistency, the bucket sizes can be easily predicted from the elimination process along the ordered graph. Consider the *moral graph* of a given Bayesian network. This graph has a node for each variable and any two variables appearing in the same CPT are connected. The moral graph of the network in Figure 4.1(a) is given in Figure 4.1(b). If we take this moral graph and impose an ordering on its nodes, the induced-width of the ordered graph of each node captures the number of variables which would be processed in that bucket. We demonstrate this next.

Example 4.4 Recall the definition of induced graph (Definition 3.7). The induced moral graph in Figure 4.8, relative to ordering $d_1 = A, C, B, F, D, G$ is depicted in Figure 4.8a. Along this ordering the induced ordered graph was not added any edges over the original graph, since all the earlier neighbors of each node are already connected. The induced width of this graph is 2. Indeed, in this case, the maximum arity of functions recorded by the algorithm is 2. For ordering $d_2 = A, F, D, C, B, G$, the ordered moral graph is depicted in Figure 4.8b and the induced graph is given in Figure 4.8c. In this ordering, the induced width is not the same as the width. For example,

A	B	C	F	$f(A,B,C,F) = P(F\|B,C) \cdot P(B\|A)$
summer	false	false	true	0.2×0.1 =0.02
summer	false	true	true	0.2×0.8 =0.16
fall	false	false	true	0.6×0.1 = 0.06
fall	false	true	true	0.6×0.8 = 0.46
winter	false	false	true	0.9×0.1 = 0.09
winter	false	true	true	0.9×0.8 = 0.72
spring	false	false	true	0.4×0.1 =0.04
spring	false	true	true	0.4×0.8 =0.32
summer	true	false	true	0.8×0.9 =0.72
summer	true	true	true	0.8×0.95 =0.76
fall	true	false	true	0.4×0.9 =0.36
fall	true	true	true	0.4×0.95 =0.38
winter	true	false	true	0.1×0.9 =0.09
winter	true	true	true	0.1×0.95 =0.095
spring	true	false	true	0.6×0.9 =0.42
spring	true	true	true	0.6×0.95 =0.57

A	C	F	$\lambda_B(A,B,F) = \sum_B f(A,B,C,F)$
summer	false	true	0.02+0.72 = 0.74
fall	false	true	0.06 + 0.36 = 0.42
winter	false	true	0.09 +0.09 = 0.18
spring	false	true	0.04 + 0.42 = 0.46
summer	true	true	0.72 + 0.16 = 0.88
fall	true	true	0.46 + 0.38 = 0.84
winter	true	true	0.72 + 0.095 = 0.815
spring	true	true	0.32 + 0.57 = 0.89

Figure 4.7: Processing the functions in the bucket of B.

the width of C is initially 2, but its induced width is 3. The maximum induced width over all the variables in this ordering is 4 which is the induced-width of this ordering.

Theorem 4.5 Complexity of BE-bel. *Given a Byaesian network whose moral graph is G, let $w^*(d)$ be its induced width of G along ordering d, k the maximum domain size, and r be the number of input CPTs. The time complexity of BE-bel is $O(r \cdot k^{w^*(d)+1})$ and its space complexity is $O(n \cdot k^{w^*(d)})$ (see Appendix for a proof).*

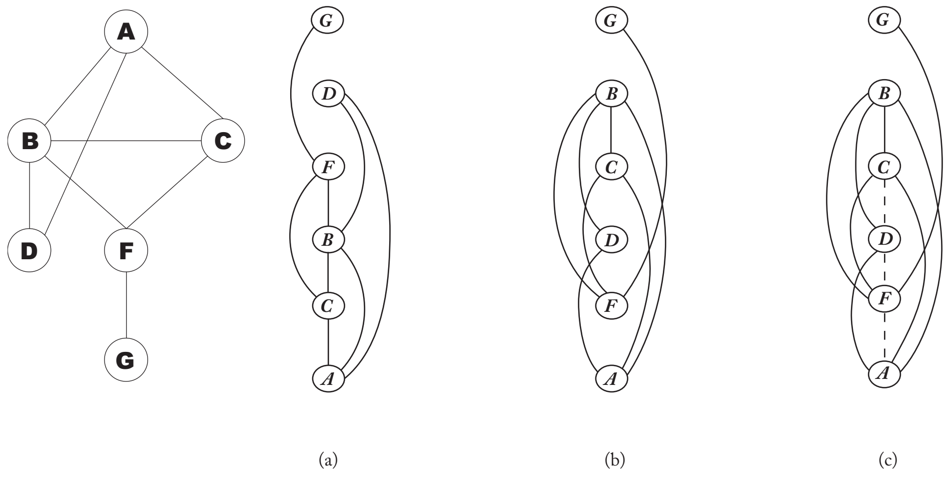


Figure 4.8: Two orderings, d_1 (a) and d_2 (b), of our example moral graph. In (c) the induced graph along ordering d_2.

4.1.3 THE IMPACT OF OBSERVATIONS

In this section we will see that observations, which are variable assignments, can have two opposing effects on the complexity of bucket-elimination BE-bel. One effect is of universal simplification and applies to any graphical model, while the other introduces complexity but is specific to likelihood queries over Bayesian networks.

Evidence removes connectivity

The presence of observed variables, which we call evidence in the Bayesian network context, is inherent to queries over probabilistic networks. From a computational perspective evidence is just an assignments of values to a subset of the variables. It turns out that the presence of such partial assignments can significantly simplify inference algorithms such as bucket elimination. In fact, we will see that this property of variable instantiations, or conditioning, as it is sometime called, is the basis for algorithms that combine search and variable-elimination, to be discussed in Chapter 7.

Take our belief network example with ordering d_1 and suppose we wish to compute the belief in A, having observed $B = b_0$. When the algorithm arrives at $bucket_B$, the bucket contains the three functions $P(b|a)$, $\lambda_D(b, a)$, and $\lambda_F(b, c)$, as well as the observation $B = b_0$ (see Figure 4.3 and add $B = b_0$ to $bucket_B$). Note that b_0 represent a specific value in the domain of B while b stands for an arbitrary value in its domain.

The processing rule dictates computing $\lambda_B(a, c) = P(b_0|a)\lambda_D(b_0, a)\lambda_F(b_0, c)$. Namely, generating and recording a two-dimensioned function. It would be more effective, however, to apply the assignment b_0 to each function in the bucket *separately* and then put the individual resulting functions into lower buckets. In other words, we can generate $\lambda_1(a) = P(b_0|a)$ and $\lambda_2(a) = \lambda_D(b_0, a)$, each of which has a single variable in its scope which will be placed in bucket A, and $\lambda_F(b_0, c)$, which will be placed in bucket C. By doing so, we avoid increasing the dimensionality of the recorded functions. In order to exploit this we introduce a special rule for processing buckets with observations (see step 3 in the algorithm): the observed value is assigned to each function in a bucket, and each function generated by this assignment is moved to the appropriate lower bucket.

Considering now ordering d_2, $bucket_B$ contains $P(b|a), P(d|b, a), P(f|c, b)$, and $B = b_0$ (see Figure 4.4a). The special rule for processing buckets holding observations will place the function $P(b_0|a)$ in $bucket_A$, $P(d|b_0, a)$ in $bucket_D$, and $P(f|c, b_0)$ in $bucket_F$. In subsequent processing only one-dimensional functions will be recorded. We see that in this case too the presence of observations reduces complexity: buckets of observed variables are processed in linear time and their recorded functions do not create functions on new subsets of variables.

Alternatively, we could just preprocess all the functions in which B appears and assign each the value b_0. This will reduce those functions scope and remove variable B altogether. We can then apply BE to the resulting pre-processed problem. Both methods will lead to an identical performance, but using an explicit rule for observations during BE allows for a more general and dynamic treatment. It can later be generalized by replacing observations by more general constraints (see Section 4.5).

In order to see the implication of the observation rule computationally, we can modify the way we manipulate the ordered moral graph and will not add arcs among parents of observed variables when computing the induced graph. This will permit a tighter bound on complexity. To capture this refinement we use the notion of *conditional induced graph*.

Definition 4.6 Conditional induced-width. Given a graph G, the conditional induced graph relative to ordering d and evidence variables $\mathbf{E}$, denoted $w^*_{\mathbf{E}}(d)$, is generated, processing the ordered graph from last to first, by connecting the earlier neighbors of unobserved nodes only. The *conditional induced width* is the width of the conditional induced graph, disregarding observed nodes.

For example, in Figure 4.9a-b we show the ordered moral graph and induced ordered moral graph of the graph in Figures 4.1a–b. In Figure 4.9c the arcs connected to the observed node B are marked by broken lines and are disregarded, resulting in the conditional induced-graph. Modifying the complexity in Theorem 4.5, we get the following.

Theorem 4.7 *Given a Bayesian network having n variables, algorithm BE-bel when using ordering d and evidence on variables $\mathbf{E} = \mathbf{e}$, is time and space exponential in the conditional induced width $w^*_{\mathbf{E}}(d)$ of the network's ordered moral graph. Specifically, its time complexity is $O(r \cdot k^{w^*_{\mathbf{E}}(d)+1})$ and its space complexity is $O(n \cdot k^{w^*_{\mathbf{E}}(d)})$.* □

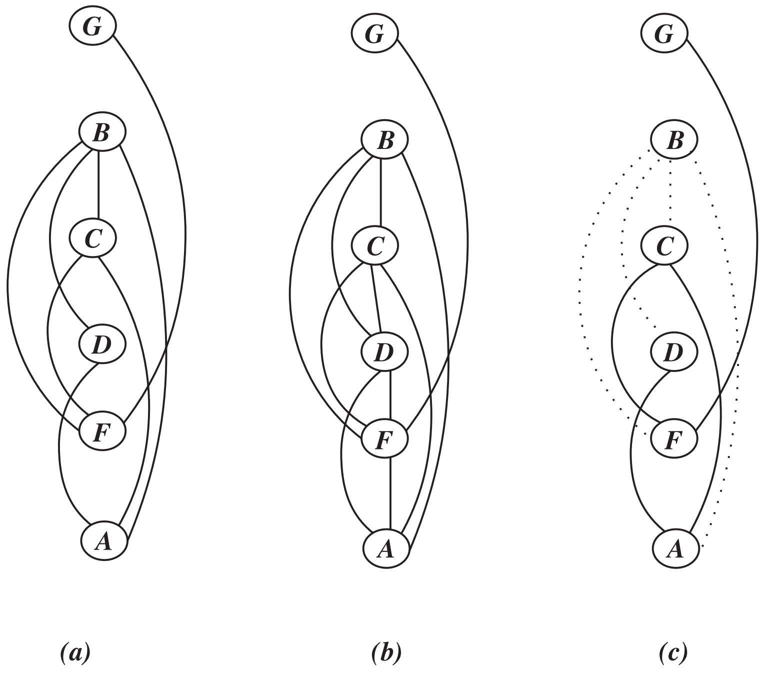


Figure 4.9: Adjusted induced graph relative to observing B.

It is easy to see that the conditional induced width is the induced width obtained by removing the evidence variables altogether (which correspond to the alternative preprocessing mentioned above).

Evidence creating connectivity: relevant subnetworks

We saw that observation can simplify computation. But, in Bayesian networks observation can also complicate inference. For example, when there is no evidence, computing the belief of a variable depends only on its non-descendant portion. This is because Bayesian networks functions are local probability distributions, and such functions, by definition, sum to 1. If we identify the portion of the network that is irrelevant, we can skip processing some of the buckets. For example, if we use a *topological ordering* from root to leaves along the directed acyclic graph (where parents precede their child nodes) and assuming that the queried variable is the first in the ordering, we can identify skippable buckets dynamically during processing.

Proposition 4.8 *Given a Bayesian network and a topological ordering $X_1, ..., X_n$, that begins a query variable X_1, algorithm BE-bel, computing $P(X_1|\mathrm{e})$, can skip a bucket if during processing the bucket contains no evidence variables and no newly computed messages.*

Proof: If topological ordering is used, each bucket of a variable X contains initially at most one function, $P(X|pa(X))$. Clearly, if there is neither evidence nor new functions in the bucket the summation operation $\sum_x P(x|pa(X))$ will yield the constant 1. □

Example 4.9 Consider again the belief network whose acyclic graph is given in Figure 4.1a and the ordering $d_1 = A, C, B, F, D, G$. Assume we want to update the belief in variable A given evidence on F. Obviously the buckets of G and D can be skipped and processing should start with $bucket_F$. Once $bucket_F$ is processed, the remaining buckets in the ordered processing are not skippable.

Alternatively, we can prune the non-relevant portion of the Bayesian network in advance, before committing to any processing ordering. The relevant subnetwork is called *ancestral subnetwork* and is defined recursively as follows.

Definition 4.10 Ancestral graph. Given a Bayesian network's directed graph $G = (X, E)$, and a query involving variables $\mathbf{S}$ (including the evidence variables), the ancestral graph of G, G_{anc}, relative to $\mathbf{S} \subseteq \mathbf{X}$, includes all variables in $\mathbf{S}$ and if a node is in G_{anc}, its parents are also in G_{anc}.

Example 4.11 Continuing with the example from Figure 4.1a, and assuming we want to assess the belief in A given evidence on F, the relevant ordered moral graph in Figures 4.1b should be modified by deleting nodes D and G. The resulting graph has nodes A, B, C, and F only.

Theorem 4.12 *Given a Bayesian* $\mathcal{B} = \langle \mathbf{X}, \mathbf{D}, \mathbf{P}_G, \prod \rangle$ *and a query* $P(\mathbf{Y}|\mathbf{e})$, *when* $\mathbf{Y} \subseteq \mathbf{X}$ *and* $\mathbf{E} \subseteq \mathbf{X}$ *is the evidence variable set, we can compute* $P(\mathbf{Y}|\mathbf{e})$ *by considering only the ancestral Bayesian network defined by* $G_{\mathbf{Y} \cup \mathbf{E}}$. *(Exercise: Prove the theorem.)*

4.2 BUCKET ELIMINATION FOR OPTIMIZATION TASKS

Belief-updating answers the question: "What is the likelihood of a variable given the observed data?" Answering that question, however, is often not enough; we want to be able to find the most likely explanation for the data we encounter. This is an optimization problem, and while we pose the problem here on a probabilistic network, it is a problem that is representative of optimization tasks on many types of graphical model. The query is called mpe.

The most probable explanation (mpe) task appears in numerous applications. Examples range from diagnosis and design of probabilistic codes to haplotype recognition in the context of family trees, and medical and circuit diagnosis. For example, given data on clinical findings, it may suggest the most likely disease a patient is suffering from. In decoding, the task is to identify the most likely input message which was transmitted over a noisy channel, given the observed output. Although the relevant task here is finding the most likely assignment over a *subset* of hypothesis variables which would correspond to a *marginal map* query, the mpe is close enough and is often used in applications (see [Darwiche, 2009] for more examples). Finally, the queries of mpe/map (see Chapter 2) drive most of the learning algorithms for graphical model [Koller and Friedman, 2009]. Our focus here is on algorithms for answering such queries on a given graphical model.

4.2.1 A BUCKET ELIMINATION ALGORITHM FOR MPE

Given a Bayesian network $\mathcal{B} = \langle \mathbf{X}, \mathbf{D}, \mathbf{P}_G, \prod \rangle$, the mpe task seeks an assignment to all the variables that has the maximal probability given the evidence. Namely, the task is to find a full instantiation $\mathbf{x}^0$ such that $P(\mathbf{x}^0) = \max_{\mathbf{x}} P(\mathbf{x}, \mathbf{e})$, where denoting $\mathbf{x} = (x_1, ..., x_n)$, $P(\mathbf{x}, \mathbf{e}) = \prod_i P(x_i, \mathbf{e}|\mathbf{x}_{pa_i})$. (Remember the $\mathbf{x}_{pa_i}$ is the assignments to $\mathbf{x}$ restricted to the variables in the parent set of X_i.) Given a variable ordering $d = X_1, ..., X_n$, we can accomplish this task by performing maximization operation, variable by variable, along the ordering from last to first (i.e., right to left), migrating to the left all CPTs that do not mention the maximizing variable. We will derive this algorithm in a similar way to that in which we derived BE-bel. Using the notation defined earlier for operations on functions, our goal is to find M, s.t.

$$M = \max_{\mathbf{x}} P(\mathbf{x}_{(1..n)}, \mathbf{e}) = \max_{\mathbf{x}_{(1..n-1)}} \max_{x_n} \prod_i P(x_i, \mathbf{e}|\mathbf{x}_{pa_i})$$

$$= \max_{\mathbf{x}_{(1..n-1)}} \prod_{X_i \in X - F_n} P(x_i, \mathbf{e}|\mathbf{x}_{pa_i}) \cdot \max_{x_n} P(x_n, \mathbf{e}|\mathbf{x}_{pa_n}) \prod_{X_i \in ch_n} P(x_i, \mathbf{e}|\mathbf{x}_{pa_i})$$

$$= \max_{\mathbf{x}_{(1..n-1)}} \prod_{X_i \in X - F_n} P(x_i, \mathbf{e}|\mathbf{x}_{pa_i}) \cdot h_n(\mathbf{x}_{S_n})$$

where

$$h_n(\mathbf{x}_{S_n}) = \max_{x_n} P(x_n, \mathbf{e}|\mathbf{x}_{pa_n}) \prod_{X_i \in ch_n} P(x_i, \mathbf{e}|\mathbf{x}_{pa_i})$$

and S_n is the scope of the generated function h_n, and F_n is the family of X_n and ch_n is the child variables of X_n. Clearly, the algebraic manipulation of the above expressions is the same as the algebraic manipulation for belief updating where summation is replaced by maximization. Consequently, the bucket elimination procedure *BE-mpe* is identical to BE-bel except for this change in the margnalization operator.

Given ordering $d = (X_1, ..., X_n)$, the conditional probability tables are partitioned as before. To process each bucket, we take the product of all the functions that reside in the bucket and then eliminate the bucket's variable by maximization. We distinguish again between the original function in the bucket whose product is denoted ψ_p and the messages, which in this case will be denoted by h. The generated function in the bucket of X_p is $h_p : S_p \rightarrow R$, $h_p = \max_{X_p} \psi_p \cdot \prod_{i=1}^{j} h_i$, where $S_p = scope(\psi_p) \cup \cup_i scope(h_i) - \{X_p\}$, is the order restricted Markov blanket and it is placed in the bucket of the largest-index variable in S_p. If the function is a constant, we can place it directly in the first bucket; constant functions are not necessary to determine the exact mpe value.

Bucket processing continues from the last to the first variable. Once all buckets are processed, the mpe value can be extracted as the maximizing product of functions in the first bucket. At this point we know the mpe value but we have not generated an optimizing tuple (also called configuration). The algorithm initiates a *forward phase* to compute an mpe tuple by assigning values to the variables along the ordering from X_1 to X_n, consulting the information recorded in each bucket. Specifically, the value x_i of X_i is selected to maximize the product in $bucket_i$ given the partial assignment $\mathbf{x}_{(1..(i-1))} = (x_1, ..., x_{i-1})$. The algorithm is presented in Figure 4.10. Observed variables are handled as in BE-bel.

```
Algorithm BE-mpe
Input: A belief network $\mathcal{B} = \langle \mathbf{X}, \mathbf{D}, \mathbf{P}_G, \prod \rangle$, where $\mathcal{P} = \{P_1, ..., P_n\}$; an ordering of the
variables, $d = X_1, ..., X_n$; observations $\mathbf{e}$.
Output: The most probable configuration given the evidence.
1. Initialize: Generate an ordered partition of the conditional probability function, $bucket_1, \ldots,$
$bucket_n$, where $bucket_i$ contains all functions whose highest variable is $X_i$. Put each observed
variable in its bucket. Let $\psi_i$ be the product of input function in a bucket and let $h_i$ be the messages
in the bucket.

2. Backward: For $p \leftarrow n$ downto 1, do
for all the functions $h_1, h_2, ..., h_j$ in $bucket_p$, do
    • If (observed variable) $bucket_p$ contains $X_p = x_p$, assign $X_p = x_p$ to each function and put
      each in appropriate bucket.
    • else, Generate functions $h_p \Leftarrow \max_{X_p} \psi_p \cdot \Pi_{i=1}^{j} h_i$
      Add $h_p$ to the bucket of the largest-index variable in $scope(h_p)$.
3. Forward:
    • Generate the mpe cost by maximizing over $X_1$, the product in $bucket_1$. Namely $mpe =$
      $max_{X_1} \psi_1 \prod_j h_{1_j}$.
    • (generate an mpe tuple)
      For $i = 1$ to $n$ along $d$ do: Given $\mathbf{x}_{(1..(i-1))} = (x_1, ..., x_{i-1})$ choose $x_i^o = argmax_{X_i} \psi_i \cdot$
      $\Pi_{\{h_j \in\, bucket_i\}} h_j(\mathbf{x}_{(1..(i-1))})$.
4. Output: mpe and configuration $\mathbf{x}^o$.
```

Figure 4.10: Algorithm *BE-mpe*.

Example 4.13 Consider again the belief network in Figure 4.1a. Given the ordering $d = A, C, B, F, D, G$ and the evidence $G = 1$, we process variables from last to first once partitioning the conditional probability functions into buckets, as was shown in Figure 4.2 To process G, assign $G = 1$, get $h_G(f) = P(G = 1|f)$ and place the result in $bucket_F$. We next process

$bucket_D$ by computing $h_D(b,a) = \max_d P(d|b,a)$ and put the result in $bucket_B$. Bucket F, which is next to be processed, now contains two functions: $P(f|b,c)$ and $h_G(f)$. We Compute $h_F(b,c) = \max_f p(f|b,c) \cdot h_G(f)$, and place the resulting function in $bucket_B$. To process $bucket_B$, we record the function $h_B(a,c) = \max_b P(b|a) \cdot h_D(b,a) \cdot h_F(b,c)$ and place it in $bucket_C$. To process C (to eliminate C), we compute $h_C(a) = \max_c P(c|a) \cdot h_B(a,c)$ and place it in $bucket_A$. Finally, the mpe value given in $bucket_A$, $M = \max_a P(a) \cdot h_C(a)$, is determined. Next, the mpe configuration is generated by going forward through the buckets. First, the value a^0 satisfying $a^0 = argmax_a P(a) h_C(a)$ is selected. Subsequently, the value of C, $c^0 = argmax_c P(c|a^0) h_B(a^0, c)$ is determined. Next, $b^0 = argmax_b P(b|a^0) h_D(b, a^0) h_F(b, c^0)$ is selected, and so on. The schematic computation is the same as in Figure 4.3 where λ is simply replaced by h.

The backward process can be viewed as a compilation phase in which we compile information regarding the most probable extension (cost to go) of partial tuples to variables higher in the ordering.

Complexity. As in the case of belief updating, the complexity of BE-mpe is bounded exponentially by the arity of the recorded functions, and those functions depend on the induced width and the evidence.

Theorem 4.14 Soundness and Complexity. *Algorithm BE-mpe is complete for the mpe task. Its time and space complexity are $O(r \cdot k^{w^*_{\mathbf{E}}(d)+1})$ and $O(n \cdot k^{w^*_{\mathbf{E}}(d)})$, respectively, where n is the number of variables, k bound the domain size and $w^*_{\mathbf{E}}(d)$ is the induced width of the ordered moral graph along d, conditioned on the evidence* $\mathbf{E}$. $\square$

4.2.2 A BUCKET ELIMINATION ALGORITHM FOR MAP

The maximum a'posteriori hypothesis map[1] task is a generalization of both mpe and belief updating. It asks for the maximal probability associated with a *subset of hypothesis variables* and is widely applicable especially for diagnosis tasks. Belief updating is the special case where the hypothesis variables are just single variables. The mpe query is the special case when the hypothesis variables are all the variables. We will see that since it is a mixture of the previous two tasks, in its bucket-elimination algorithm some of the variables are eliminated by summation while others by maximization.

Given a Bayesian network, a subset of hypothesized variables $A = \{A_1, ..., A_k\}$, and some evidence e, the problem is to find an assignment (i.e., configuration) to A having maximum probability given the evidence compared with all other assignments to A. Namely, the task is to find $a^o = \text{argmax}_{a_1,...,a_k} P(a_1, ..., a_k, \mathbf{e})$ (see also Definition 2.22). So, we wish to compute $\max_{\mathbf{a}_{1..k}} P(a_1, ..., a_k, e) = \max_{\mathbf{a}_{1..k}} \sum_{\mathbf{x}_{(k+1..n)}} \prod_{i=1}^n P(x_i, \mathbf{e}|\mathbf{x}_{pa_i})$ where $\mathbf{x} = (a_1, ..., a_k, x_{k+1}, ..., x_n)$. Algorithm *BE-map* in Figure 4.11 considers only orderings in which the hypothesized variables start the ordering because summation should be applied first to the subset

[1] Sometimes map is meant to refer to the mpe, and the map task is called marginal map.

of variables which are in $\mathbf{X} - \mathbf{A}$, and subsequently maximization is applied to the variables in $\mathbf{A}$. Since summation and maximization cannot be permuted we have to be restricted in the orderings. Like BE-mpe, the algorithm has a backward phase and a forward phase, but the forward phase is restricted to the hypothesized variables only. Because only restricted orderings are allowed, the algorithm may be forced to have far higher induced-width than would otherwise be allowed.

```
Algorithm BE-map
Input: A Bayesian network $\mathcal{B} = \langle \mathbf{X}, \mathbf{D}, \mathbf{P}_G, \prod \rangle$, $P = \{P_1, ..., P_n\}$; a subset of hypothesis variables $A = \{A_1, ..., A_k\}$; an ordering of the variables, $d$, in which the $A$'s are first in the ordering; observations $e$. $\psi_i$ is the product of input function in the bucket of $X_i$.
Output: A most probable assignment $A = a$.
1. Initialize: Generate an ordered partition of the conditional probability functions, $bucket_1, \ldots, bucket_n$, where $bucket_i$ contains all functions whose highest variable is $X_i$.
2. Backwards For $p \leftarrow n$ downto 1, do
for all the message functions $\beta_1, \beta_2, ..., \beta_j$ in $bucket_p$ and for $\psi_p$ do
    • If (observed variable) $bucket_p$ contains the observation $X_p = x_p$, assign $X_p = x_p$ to each $\beta_i$ and $\psi_p$ and put each in appropriate bucket.
    • else, If $X_p$ is not in $A$, then $\beta_p \Leftarrow \sum_{X_p} \psi_p \cdot \Pi_{i=1}^{j} \beta_i$;
      else, $(X_p \in A)$, $\beta_p \Leftarrow \max_{X_p} \psi_p \cdot \prod_{i=1}^{j} \beta_i$
      Place $\beta_p$ in the bucket of the largest-index variable in $scope(\beta_p)$.
3. Forward: Assign values, in the ordering $d = A_1, ..., A_k$, using the information recorded in each bucket in a similar way to the forward pass in BE-mpe.
4. Output: Map and the corresponding configuration over $A$.
```

Figure 4.11: Algorithm *BE-map*.

Theorem 4.15 *Algorithm BE-map is complete for the map task for orderings started by the hypothesis variables. Its time and space complexity are $O(r \cdot k^{w^*_{\mathbf{E}}(d)+1})$ and $O(n \cdot k^{w^*_{\mathbf{E}}(d)})$, respectively, where n is the number of variables in graph, k bounds the domain size and $w^*_{\mathbf{E}}(d)$ is the conditioned induced width of its moral graph along a restricted ordering d, relative to evidence variables* $\mathbf{E}$. *(Prove as an exercise.)* $\square$

4.3 BUCKET ELIMINATION FOR MARKOV NETWORKS

Recalling Definition 2.23 of a Markov network which is presented here for convenience.

Definition 4.16 Markov networks. A Markov network is a graphical model $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{H}, \prod \rangle$ where $\mathbf{H} = \{\psi_1, \ldots, \psi_m\}$ is a set of potential functions where each potential ψ_i is a non-negative real-valued function defined over a scope of variables $\mathbf{S}_i$. The Markov network represents a global joint distribution over the variables $\mathbf{X}$ given by:

$$P(\mathbf{x}) = \frac{1}{Z}\prod_{i=1}^{m}\psi_i(\mathbf{x}) \quad , \quad Z = \sum_{\mathbf{x}}\prod_{i=1}^{m}\psi_i(\mathbf{x})$$

where the normalizing constant Z is referred to as the partition function.

It is easy to see that the bucket-elimination algorithms we presented for Bayesian networks are immediately applicable to all the main queries over Markov networks. All we need to do is replace the input conditional probabilities through which a Bayesian network is specified by the collection of local potential functions or factors denoted by $\psi(.)$. The query of computing posterior marginals is accomplished by BE-bel, computing mpe and map are accomplished by BE-mpe and BE-map, respectively. Since the partition function is identical, mathematically to the expression of probability of evidence, the task of computing Z is identical algorithmically to the task of computing the probability of the evidence.

4.4 BUCKET ELIMINATION FOR COST NETWORKS AND DYNAMIC PROGRAMMING

As we mentioned at the outset, bucket-elimination algorithms are variations of a very well known class of optimization algorithms known as *Dynamic Programming* [Bellman, 1957, Bertele and Brioschi, 1972]. Here we make the connection explicit, observing that BE-mpe is a dynamic programming scheme with some simple transformations.

That BE-mpe is dynamic programming becomes apparent once we transform the mpe's cost function, which has a product combination operator, into the traditional additive combination operator using the log function. For example,

$$P(a,b,c,d,f,g) = P(a)P(b|a)P(c|a)P(f|b,c)P(d|a,b)P(g|f)$$

becomes

$$C(a,b,c,d,e) = -logP = C(a) + C(b,a) + C(c,a) + C(f,b,c) + C(d,a,b) + C(g,f)$$

where each $C_i \;=\; -\,logP_i$.

The general dynamic programming algorithm is defined over *cost networks* (see Section 2.4). As we showed a *cost network* is a tuple $\mathcal{C} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \sum \rangle$, where $\mathbf{X} = \{X_1, ..., X_n\}$ are variables over domains $\mathbf{D} = \{D_1, ..., D_n\}$, F is a set of real-valued cost functions $C_1, ..., C_l$, defined over scopes $S_1, ..., S_l$. The task is to find an assignment or a configuration to all the variables that minimizes the global function $\sum_i C_i$.

A straightforward elimination process similar to that of BE-mpe (where the product is replaced by summation and maximization by minimization) yields the non-serial dynamic programming algorithm in [Bertele and Brioschi, 1972]. The algorithm, called here *BE-opt*, is given in Figure 4.12. Evidence is not assumed to be part of the input, so this part of the algorithm is omitted.

A schematic execution of our example along ordering $d = G, A, F, D, C, B$ is depicted in Figure 4.13. It is identical to what we saw for BE-mpe, except that the generated functions are computed by min-sum, instead of max-product. Not surprisingly, we can show the following.

```
Algorithm BE-opt
Input: A cost network $\mathcal{C} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \sum \rangle$, $F = \{C_1, ..., C_l\}$; ordering $d$.
Output: A minimal cost assignment.
1. Initialize: Partition the cost components into buckets. Define $\psi_i$ as the sum of the input cost functions in bucket $X_i$.
2. Process buckets from $p \leftarrow n$ downto 1
For $\psi_p$ and the cost messages $h_1, h_2, ..., h_j$ in $bucket_p$, do:
    • (sum and minimize):
      $h_p \Leftarrow min_{X_p}(\psi_p + \sum_{i=1}^{j} h_i)$. Place $h_p$ into the largest index variable in its scope.
3. Forward: Assign minimizing values in ordering $d$, consulting functions in each bucket (as in BE-mpe).
4. Return: Optimal cost and an optimizing assignment.
```

Figure 4.12: Dynamic programming as BE-opt.

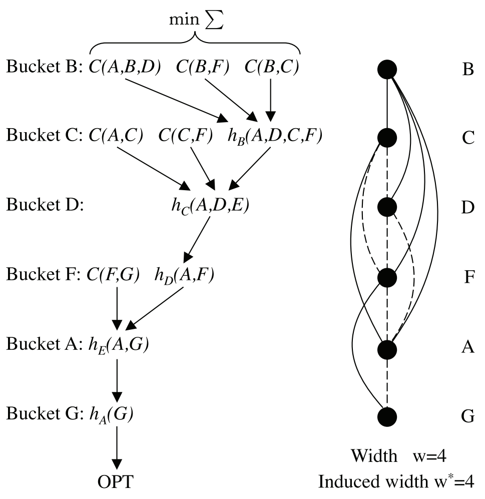


Figure 4.13: Schematic execution of BE-opt.

Theorem 4.17 *Given a cost network $\mathcal{C} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \sum \rangle$, BE-opt is complete for finding an optimal cost solution. Its time and space complexity are $O(r \cdot k^{w^*(d)+1})$ and $O(n \cdot k^{w^*(d)})$, respectively, where n is the number of variables in graph, k bounds the domain size, and $w^*(d)$ is the induced width of its primal graph along d.* □

Consulting again the various classes of cost networks elaborated up on in Section 2.4, algorithm, BE-opt is applicable to all including weighted-csps, max-csps and max-sat.

4.5 BUCKET ELIMINATION FOR MIXED NETWORKS

The last class of graphical models we will address is mixed network defined in Section 2.6. To refresh, these models allow the explicit representation of both probabilistic information and constraints. The mixed network is defined by a pair of a Bayesian network and a constraint network. This pair expresses a probability distribution over all the variables which is conditioned on the requirement that all the assignments having non-zero probability satisfy all the constraints.

We will focus only on the task that is unique to this graphical model, the *constraint probability evaluation* (CPE), which can also stand for *CNF probability evaluation*. Given a mixed network $\mathcal{M}_{(\mathcal{B},\varphi)}$, where φ is a CNF formula defined on perhaps a subset of propositional variables and $\mathcal{B}$ is a Bayesian network, the CPE task is to compute $P(\mathbf{x} \in Mod(\varphi))$, where the models of φ are denoted by $Mod(\varphi)$ and are the assignments to all the variables satisfying the formula φ. We denote by $P_{\mathcal{B}}(\varphi)$ the probability that $P_{\mathcal{B}}(\mathbf{x} \in Mod(\varphi))$. We denote by $\mathbf{X}_\varphi$ the set of variables in the scope of φ. By definition,

$$P_{\mathcal{B}}(\varphi) = \sum_{\mathbf{x}_\varphi \in Mod(\varphi)} P(\mathbf{x}_\varphi)$$

Using the belief network product form we get:

$$P_{\mathcal{B}}(\varphi) = \sum_{\{\mathbf{x} | \mathbf{x}_\varphi \in Mod(\varphi)\}} \prod_{i=1}^{n} P(x_i | \mathbf{x}_{pa_i}).$$

We separate the summation over X_n and the rest of the variables $\mathbf{X} - \{X_n\}$ as usual and denote by φ_{+X_n} the set of all clauses defined on X_n (there may be none if it is not a proposition in φ) and by φ_{-X_n} all the rest of the clauses which are not defined on X_n. We get (denoting $P(X_i | \mathbf{x}_{pa_i})$ by P_i):

$$P_{\mathcal{B}}(\varphi) = \sum_{\{\mathbf{x}_{(1..n-1)} | \mathbf{x}_{\varphi_{-X_n}} \in Mod(\varphi_{-X_n})\}} \sum_{\{x_n | \mathbf{x}_{\varphi_{+X_n}} \in Mod(\varphi_{+X_n})\}} \prod_{i=1}^{n} P(x_i | \mathbf{x}_{pa_i}).$$

Let t_n be the set of indices of functions in the product that *do not* mention X_n, namely, are not in φ_{+X_n} and by $l_n = \{1, \ldots, n\} \setminus t_n$ we get:

$$P_{\mathcal{B}}(\varphi) = \sum_{\{\mathbf{x}_{(1..n-1)} | \mathbf{x}_{\varphi_{-X_n}} \in Mod(\varphi_{-X_n})\}} \prod_{j \in t_n} P_j \cdot \sum_{\{x_n | \mathbf{x}_{\varphi_{+X_n}} \in Mod(\varphi_{+X_n})\}} \prod_{j \in l_n} P_j.$$

Therefore:

$$P_{\mathcal{B}}(\varphi) = \sum_{\{\mathbf{x}_{(1..n-1)} | \mathbf{x}_{\varphi_{-X_n}} \in Mod(\varphi_{-X_n})\}} (\prod_{j \in t_n} P_j) \cdot \lambda_{X_n},$$

where λ_{X_n} is defined over $U_n = scope(\varphi_{+X_n})$, by

$$\lambda_{X_n} = \sum_{\{x_n | \mathbf{x}_{U_n} \in Mod(\varphi_{+X_n})\}} \prod_{j \in l_n} P_j . \tag{4.8}$$

The case of observed variables. When X_n is observed, that is constrained by a literal, the summation operation reduces to assigning the observed value to each of its CPTs *and* to each of the relevant clauses. In this case, Eq. (4.8) becomes (assume $X_n = x_n$ and $P_{(=x_n)}$ is the function instantiated by assigning x_n to X_n):

$$\lambda_{X_n} = \prod_{j \in l_n} P_{j\,(=x_n)}, \quad if\ \mathbf{x}_{U_n} \in Mod(\varphi_{+X_n}) \wedge (X_n = x_n)). \tag{4.9}$$

Otherwise, $\lambda_{X_n} = 0$. Since a tuple $\mathbf{x}_{U_n}$ satisfies $\varphi_{+X_n} \wedge (X_n = x_n)$ only if $\mathbf{x}_{U_n - X_n}$ satisfies the resolvent clause $\gamma_n = resolve(\varphi_{+X_n}, (X_n = x_n))$, we get:

$$\lambda_{X_n} = \prod_{j \in l_n} P_{j\,(=x_n)}, \ \ \textit{if}\ \mathbf{x}_{(U_n - X_n)} \in Mod(\gamma_n). \tag{4.10}$$

We can, therefore, extend the case of observed variable in a natural way: CPTs are assigned the observed value as usual while clauses are individually resolved with the unit clause $(X_n = x_n)$, and both are moved to appropriate lower buckets. This yields the following.

To Initialise, place all CPTs and clauses mentioning X_n in its bucket and then compute the function in Eq. (4.8). The computation of the rest of the expression proceeds with X_{n-1} in the same manner. This yields algorithm BE-CPE described in Figure 1 and Procedure `Process-bucket`$_p$. The elimination operation is summation for the current query. Thus, for every ordering of the propositions, once all the CPTs and clauses are partitioned, we process the buckets from last to first, in each applying the following operation. Let $\lambda_1, ...\lambda_t$ be the probabilistic functions in $bucket_P$ and $\varphi = \{\alpha_1, ...\alpha_r\}$ be the clauses. The algorithm computes a new function λ_P over $S_p = scope(\lambda_1, ...\lambda_t) \cup scope(\alpha_1, ...\alpha_r) - \{X_p\}$ defined by:

$$\lambda_P = \sum_{\{x_p | \mathbf{x}_\varphi \in Mod(\alpha_1, ..., \alpha_r)\}} \prod_j \lambda_j \ .$$

Example 4.18 Consider the belief network in Figure 4.14, which is similar to the one in Figure 2.5, and the query $\varphi = (B \vee C) \wedge (G \vee D) \wedge (\neg D \vee \neg B)$. The initial partitioning into buckets along the ordering $d = A, C, B, D, F, G$, as well as the output buckets are given in Figure 4.15. We compute:

In Bucket G: $\lambda_G(f, d) = \sum_{\{g | g \vee d = true\}} P(g|f)$
In $Bucket_F$: $\lambda_F(b, c, d) = \sum_f P(f|b, c)\lambda_G(f, d)$
In $Bucket_D$: $\lambda_D(a, b, c) = \sum_{\{d | \neg d \vee \neg b = true\}} P(d|a, b)\lambda_F(b, c, d)$

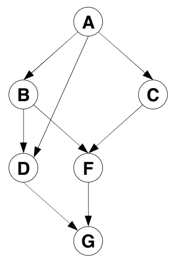
```
Algorithm 1: BE-cpe
    Input: A belief network $\mathcal{M} = (\mathcal{B}, \varphi)$, $\mathcal{B} = \langle \mathbf{X}, \mathbf{D}, \mathbf{P}_G, \prod \rangle$, where $\mathcal{B} = \{P_1, ..., P_n\}$; a
           CNF formula on $k$ propositions $\varphi = \{\alpha_1, ...\alpha_m\}$ defined over $k$ propositions; an
           ordering of the variables, $d = \{X_1, \ldots, X_n\}$.
    Output: The belief $P(\varphi)$.
  1 Place buckets with unit clauses last in the ordering (to be processed first).
    // Initialize
    Partition $\mathcal{B}$ and $\varphi$ into $bucket_1, \ldots, bucket_n$, where $bucket_i$ contains all the CPTs and
    clauses whose highest variable is $X_i$.
    Put each observed variable into its appropriate bucket. (We denote probabilistic functions
    by $\lambda$s and clauses by $\alpha$s).
  2 for $p \leftarrow n$ downto $1$ do                                          // Backward
        Let $\lambda_1, \ldots, \lambda_j$ be the functions and $\alpha_1, \ldots, \alpha_r$ be the clauses in $bucket_p$
        Process-bucket$_p$($\sum$, $(\lambda_1, \ldots, \lambda_j)$,$(\alpha_1, \ldots, \alpha_r)$)
  3 return $P(\varphi)$ as the result of processing $bucket_1$.
```

```
Procedure Process-bucket$_p$ ($\sum$, $(\lambda_1, \ldots, \lambda_j)$,$(\alpha_1, \ldots, \alpha_r)$ ).
    if $bucket_p$ contains evidence $X_p = x_p$ then
        1. Assign $X_p = x_p$ to each $\lambda_i$ and put each resulting function in the bucket of its
        latest variable
        2. Resolve each $\alpha_i$ with the unit clause, put non-tautology resolvents in the buckets
        of their latest variable and move any bucket with unit clause to top of processing
    else
        $\lambda_p \leftarrow \sum_{\{x_p | \mathbf{x}_{U_p} \in Mod(\alpha_1, \ldots, \alpha_r)\}} \prod_{i=1}^{j} \lambda_i$
        Add $\lambda_p$ to the bucket of the latest variable in $S_p$, where
        $S_p = scope(\lambda_1, ..., \lambda_j, \alpha_1, ..., \alpha_r)$, $U_p = scope(\alpha_1, ..., \alpha_r)$.
```

In $Bucket_B$: $\lambda_B(a, c) = \sum_{\{b | b \vee c = true\}} P(b|a) \lambda_D(a, b, c) \lambda_F(b, c)$
In $Bucket_C$: $\lambda_C(a) = \sum_c P(c|a) \lambda_B(a, c)$
In $Bucket_A$: $\lambda_A = \sum_a P(a) \lambda_C(a)$
$P(\varphi) = \lambda_A$.

For example in $bucket_G$, $\lambda_G(f, d = 0) = P(g = 1|f)$, because if $D = 0$ g must get the value "1", while $\lambda_G(f, d = 1) = P(g = 0|f) + P(g = 1|f)$. In summary, we have the following.

Theorem 4.19 Correctness and completeness. *Algorithm BE-cpe is sound and complete for the CPE task.*



(a) Directed acyclic graph (b) Moral graph

Figure 4.14: Belief network.

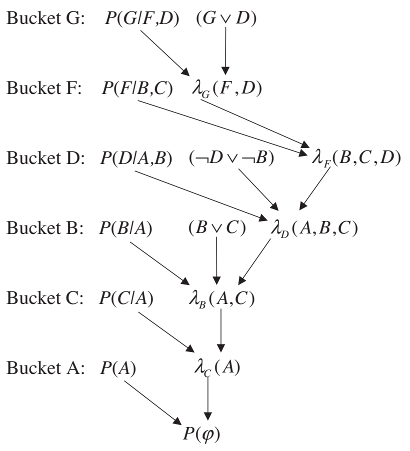

Bucket G: P(G|F,D) (G ∨ D)
Bucket F: P(F|B,C) λ_G(F,D)
Bucket D: P(D|A,B) (¬D ∨ ¬B) λ_F(B,C,D)
Bucket B: P(B|A) (B ∨ C) λ_D(A,B,C)
Bucket C: P(C|A) λ_B(A,C)
Bucket A: P(A) λ_C(A)
P(φ)

Figure 4.15: Execution of BE-CPE.

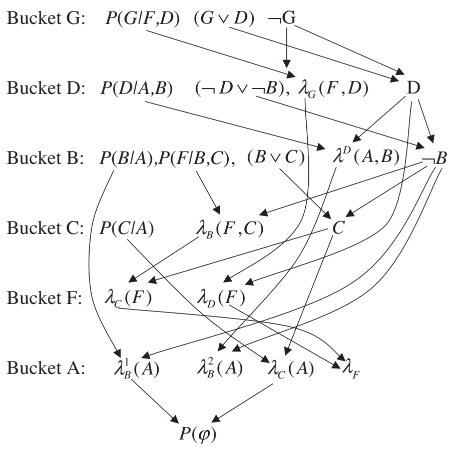

Bucket G: P(G|F,D) (G ∨ D) ¬G
Bucket D: P(D|A,B) (¬D ∨ ¬B), λ_G(F,D) D
Bucket B: P(B|A),P(F|B,C), (B ∨ C) λ^D(A,B) ¬B
Bucket C: P(C|A) λ_B(F,C) C
Bucket F: λ_C(F) λ_D(F)
Bucket A: λ_B^1(A) λ_B^2(A) λ_C(A) λ_F
P(φ)

Figure 4.16: Execution of BE-CPE (evidence $\neg G$).

It is easy to see that the complexity of the algorithm depends on the mixed graph which is the union of the moral graph and the constraint graph, in the usual way.

Theorem 4.20 Complexity of BE-cpe. *Given a mixed network $M_{\mathcal{B},\varphi}$ having mixed graph is G, with $w^*(d)$ its induced width along ordering d, k the maximum domain size and r be the number of input functions. The time complexity of BE-cpe is $O(r \cdot k^{w^*(d)+1})$ and its space complexity is $O(n \cdot k^{w^*(d)})$. (Prove as an exercise.)*

Notice that algorithm BE-cpe also includes a unit resolution step whenever possible (see Procedure `Process-bucket`$_p$) and a dynamic reordering of the buckets that prefers processing buckets that include unit clauses. This may have a significant impact on efficiency because treating observations (namely unit clauses) in a special way can avoid creating new dependencies as we already observed.

Example 4.21 Let's now extend the example by adding $\neg G$ to the query. This will place $\neg G$ in the bucket of G. When processing bucket G, unit resolution creates the unit clause D, which is then placed in bucket D. Next, processing bucket F creates a probabilistic function on the two variables B and C. Processing bucket D that now contains a unit clause will assign the value $D = 1$ to the CPT in that bucket and apply unit resolution, generating the unit clause $\neg B$ that is placed in bucket B. Subsequently, in bucket B we can apply unit resolution again, generating C placed in bucket C, and so on. In other words, aside from bucket F, we were able to process all buckets as observed buckets, by propagating the observations (see Figure 4.16.) To incorporate dynamic variable ordering, after processing bucket G, we move bucket D to the top of the processing list (since it has a unit clause). Then, following its processing, we process bucket B and then bucket C, then F, and finally A.

Since unit resolution increases the number of buckets having unit clauses, and since those are processed in linear time, it can improve performance substantially. Such buckets can be identified a priori by applying unit resolution on the CNF formula or arc-consistency if we have a constraint expression. In fact, any level of resolution can be applied in each bucket. This can yield stronger CNF expressions in each bucket and may help improve the computation of the probabilistic functions (see [Dechter, 2003]).

4.6 THE GENERAL BUCKET ELIMINATION

We now summarize and generalize the bucket elimination algorithm using the two operators of *combination* and *marginalization*. As presented in Chapter 2, the general task can be defined over a graphical model $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \otimes \rangle$, where: $X = \{X_1, ..., X_n\}$ is a set of variables having domain of values $D = \{D_1, ..., D_n\}$ and $F = \{f_1, ..., f_k\}$ is a set of functions, where each f_i is defined over $S_i = scope(f_i)$. Given a function h and given $Y \subseteq scope(h)$, the (generalized) projection operator $\Downarrow_Y h$, $\Downarrow_Y h \in \{max_{S-Y} h, min_{S-Y} h, \pi_Y h, \sum_{S-Y} h\}$ and the (generalized) combination operator $\otimes_j f_j$ defined over $U = \cup_j scope(f_j)$, $\otimes_{j=1}^k f_j \in \{\Pi_{j=1}^k f_j, \ \sum_{j=1}^k f_j, \ \ \bowtie_j f_j\}$. All queries require computing $\Downarrow_Y \otimes_{i=1}^n f_i$. Such problems can be solved by a general bucket-elimination algorithm stated in Figure 4.17. For example, BE-bel is obtained when $\Downarrow_Y = \sum_{S-Y}$ and $\otimes_j = \Pi_j$, BE-mpe is obtained when $\Downarrow_Y = \max_{S-Y}$ and $\otimes_j = \Pi_j$, and adaptive consistency corresponds to $\Downarrow_Y = \pi_Y$ and $\otimes_j = \bowtie_j$. Similarly, Fourier elimination and directional resolution can be shown to be expressible in terms of such operators. For mixed networks the combination and marginalization are also well defined.

We will state briefly the properties of GBE.

Theorem 4.22 Correctness and complexity. *Algorithm GBE is sound and complete for its task. Its time and space complexities is exponential in the $w^*(d) + 1$ and $w^*(d)$, respectively, along the order of processing d.*

Algorithm General bucket elimination (GBE)

Input: $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \otimes \rangle$. $F = \{f_1, ..., f_n\}$ an ordering of the variables, $d = X_1, ..., X_n$; $\mathbf{Y} \subseteq \mathbf{X}$.

Output: A new compiled set of functions from which the query $\Downarrow_Y \otimes_{i=1}^n f_i$ can be derived in linear time.

```
1. Initialize: Generate an ordered partition of the functions into $bucket_1, ..., bucket_n$, where
$bucket_i$ contains all the functions whose highest variable in their scope is $X_i$. An input function
in each bucket $\psi_i$, $\psi_i = \otimes_{i=1}^n f_i$.
2. Backward: For $p \leftarrow n$ downto 1, do
for all the functions $\psi_p, \lambda_1, \lambda_2, ..., \lambda_j$ in $bucket_p$, do
    • If (observed variable) $X_p = x_p$ appears in $bucket_p$, assign $X_p = x_p$ in $\psi_p$ and to each $\lambda_i$
      and put each resulting function in appropriate bucket.
    • else, (combine and marginalize)
      $\lambda_p \leftarrow \Downarrow_{S_p} \psi_p \otimes (\otimes_{i=1}^j \lambda_i)$ and add $\lambda_p$ to the largest-index variable in $scope(\lambda_p)$.
3. Return: all the functions in each bucket.
```

Figure 4.17: Algorithm *General bucket elimination*.

4.7 SUMMARY AND BIBLIOGRAPHICAL NOTES

In the last two chapters, we showed how the bucket-elimination framework can be used to unify variable-elimination algorithms for both deterministic and probabilistic graphical models for various tasks. The algorithms take advantage of the structure of the graph. Most bucket-elimination algorithms are time and space exponential in the induced width of the underlying dependency primal graph of the problem.

Chapter 4 is based on Dechter's Bucket-elimination algorithm that appeared in [Dechter, 1999]. Among the early variable elimination algorithms we find the peeling algorithm for genetic trees [Cannings *et al.*, 1978], Zhang and Poole's VE1 algorithm [Zhang and Poole, 1996], and SPI algorithm by D'Ambrosio et al., [R.D. Shachter and Favero, 1990] which preceded both BE-bel and VE1 and provided the principle ideas in the context of belief updating. Decimation algorithms in statistical physics are also related and were applied to Boltzmann trees [Saul and Jordan, 1994].

In [R. Dechter and Pearl, 1990] the connection between optimization and constraint satisfaction and its relationship to dynamic programming is explicated. In the work of [Mitten, 1964, Shenoy, 1992] and later in [Bistarelli *et al.*, 1997] an axiomatic framework that characterize tasks that can be solved polynomially over hyper-trees, is introduced.

4.8 APPENDIX: PROOFS

Proof of Theorem 4.5

During BE-bel, each bucket creates a λ function which can be viewed as a message that it sends to a *parent* bucket, down the ordering (recall that we process the variables from last to first). Since to compute this function over w^* variables the algorithm needs to consider all the tuples defined on all the variables in the bucket, whose number is bounded by $w^* + 1$, the time to compute the function is bounded by k^{w^*+1}, and its size is bounded by k^{w^*}. For each of these k^{w^*+1} tuple we need to compute its value by considering information from each of the functions in the buckets. If r_i is the number of the bucket's original messages and deg_i is the number of messages it receives from its children, then the computation of the bucket's function is $O((r_i + deg_i + 1)k^{w^*+1})$. Therefore, summing over all the buckets, the algorithm's computation is bounded by

$$\sum_i (r_i + deg_i - 1) \cdot k^{w^*+1}.$$

We can argue that $\sum_i deg_i \leq n$, when n is the number of variables, because only a single function is generated in each bucket, and there are total of n buckets. Therefore, the total complexity can be bound by $O((r + n) \cdot k^{w^*+1})$. Assuming $r > n$, this becomes $O(r \cdot k^{w^*+1})$. The size of each λ message is $O(k^{w^*})$. Since the total number of λ messages is bounded by n, the total space complexity is $O(n \cdot k^{w^*})$. $\square$

CHAPTER 5

Tree-Clustering Schemes

In this chapter, we take the bucket elimination algorithm a step further. We will show that bucket elimination can be viewed as an algorithm that sends messages along a tree (the bucket tree). The algorithm can then be augmented with a second set of messages passed from bottom to top, yielding a message-passing schemes that belongs to the class of *cluster tree elimination* algorithms.

These latter methods have received different names in different research areas, such as join-tree clustering or junction-tree algorithms, clique-tree clustering, and hyper-tree decompositions. We will refer to all these as cluster-tree processing schemes over *tree-decompositions*. Our algorithms are applicable to a general reasoning problem described by $\mathcal{P} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \bigotimes, \Downarrow \rangle$, where the first four elements identify the graphical model and the fifth identifies the reasoning task (see Definition 2.2).

Important: we will assume the specific of probabilistic networks when developing the algorithms and the reader can just make the appropriate generalization. Also we will allow abuse of notation when including in a model its query operator whenever relevant. Henceforth we assume $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \prod, \sum \rangle$.

5.1 BUCKET-TREE ELIMINATION

The bucket-elimination algorithm, *BE-bel* (see Figure 4.5) for belief updating is designed to compute the belief of the first node in a given ordering, and the probability of evidence. However, it is often desirable to answer the belief query for each and every variable in the network. A brute-force approach will require running *BE-bel* n times, each time with a different variable at the start of the ordering. We will show next that this is unnecessary. By viewing bucket-elimination as a message passing algorithm along a rooted *bucket tree*, we can augment it with a second message passing phase in the opposite direction, from root to leaves, achieving the same goal.

Example 5.1 Consider our ongoing Bayesian network example defined over the directed acyclic graph (DAG) in Figure 4.1 and appearing again here in Figure 5.8(a). Figure 5.1a recaps the initial buckets along ordering $d = (A, B, C, D, F, G)$ and the messages, labeled by λ, that will be passed by BE from top to bottom. Figure 5.1b depicts the same computation as message-passing along a tree which we will refer to as a *bucket tree*. Notice that, as before, the ordering is displayed from the bottom up (A, the first variable, is at the bottom and G, the last one, is at the top), and the messages are passed top down. This computation results in the belief in A, $bel(A) = P(A|G = 1)$, and consulted only functions that reside in the bucket of A. What if we now want to compute $bel(D)$?

We can start the algorithm using a new ordering such as (D, A, B, C, F, G). Alternatively, rather then doing all the computations from scratch using a different variable ordering whose first variable is D, we can take the bucket tree and re-orient the edges to make D the root of the tree. Reorienting the tree so that D is the root, requires reversing only 2 edges, (B, D) and (A, B), suggesting that we only need to recompute messages from node A to B and from B to D. We can think about a new virtual partial order expressed as $(\{D, A, B\}, C, F, G)$, namely, collapsing the buckets of B, A and D into a single bucket and therefore ignoring their internal order. By definition, we can compute the belief in D by the expression

$$bel(d) = \alpha \sum_a \sum_b P(a) \cdot p(b|a) \cdot P(d|a, b) \cdot \lambda_{C \to B}(b) \ . \tag{5.1}$$

Likewise, we can also compute the belief in B by

$$bel(b) = \alpha \sum_a \sum_d P(a) \cdot p(b|a) \cdot P(d|a, b) \cdot \lambda_{C \to B}(b) \ . \tag{5.2}$$

This computation can be carried also over the bucket tree, whose downward messages were already passed, in three steps. The first executed in bucket A, where the function $P(A)$ is moved to $bucket_B$, the second is executed by $bucket_B$, computing a function (a product) that is moved to $bucket_D$. The final computation is carried in $bucket_D$. Denoting the new reverse messages by π, a new $\pi_{A \to B}(a) = P(A)$, is passed from $bucket_A$ to $bucket_B$. Then, an intermediate function is computed in $bucket_B$, to be sent to $bucket_D$, using the messages received from $bucket_C$ and $bucket_A$ and its own function,

$$\pi_{B \to D}(a, b) = p(b|a) \cdot \pi_{A \to B}(a) \cdot \lambda_{C \to B}(b) \ .$$

Finally, the belief is computed in $bucket_D$ using its current function content by

$$bel(d) = \alpha \sum_{a,b} P(d|a, b) \cdot \pi_{B \to D}(a, b). \tag{5.3}$$

This accomplishes the computation of the algebraic expression in Eq (5.1). You can see some of these messages depicted in Figure 5.2a. The belief in B can also be computed in $bucket_D$. However, if we want each bucket to compute its own belief, and since $bucket_D$ sends $P(D|A, B)$ to $bucket_B$ anyway, as part of BE, the computation of Eq. (5.2) can be carried out there, autonomously.

The example generalizes. We can compute the belief for every variable by a second message passing from the root to the leaves along the bucket tree, such that at termination the belief for each variable can be computed locally, in each bucket, consulting only the functions in its own bucket.

Let $\mathcal{M}$ be a graphical model $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \prod \rangle$ and d an ordering of its variables $X_1, ..., X_n$. Let $B_{X_1}, ..., B_{X_n}$ denote a set of buckets, one for each variable. We will use B_{X_i} and B_i interchangeably. As before, each bucket B_i contains those functions in F whose latest variable

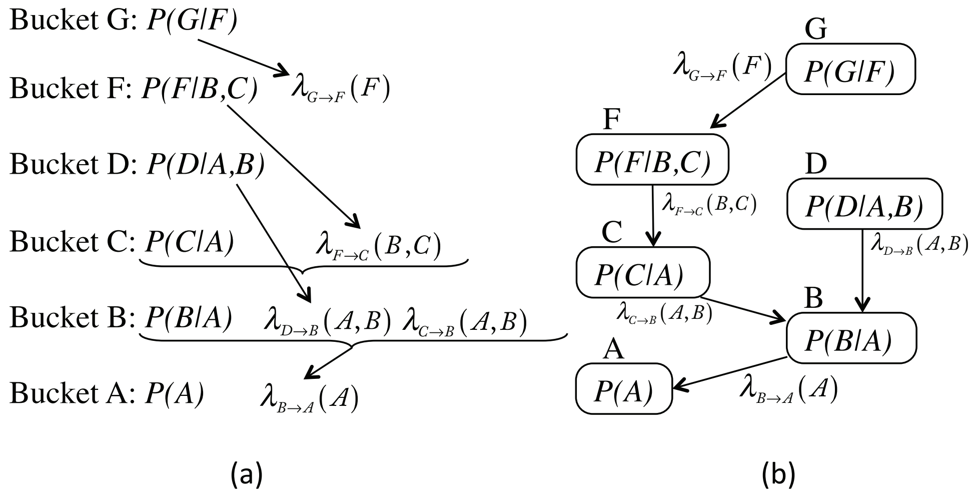

Bucket G: P(G|F)
Bucket F: P(F|B,C) $\lambda_{G\to F}(F)$
Bucket D: P(D|A,B)
Bucket C: P(C|A) $\lambda_{F\to C}(B,C)$
Bucket B: P(B|A) $\lambda_{D\to B}(A,B)$ $\lambda_{C\to B}(A,B)$
Bucket A: P(A) $\lambda_{B\to A}(A)$
(a)
G P(G|F) $\lambda_{G\to F}(F)$
F P(F|B,C) $\lambda_{F\to C}(B,C)$
D P(D|A,B) $\lambda_{D\to B}(A,B)$
C P(C|A) $\lambda_{C\to B}(A,B)$
B P(B|A) $\lambda_{B\to A}(A)$
A P(A)
(b)

Figure 5.1: Execution of BE along the bucket tree.

in ordering d is X_i (i.e., according to the bucket-partitioning rule), and, as before, we denote by ψ_i the product of functions in B_i. A bucket tree of a model $\mathcal{M}$ along d has buckets as its nodes. Bucket B_X is connected to bucket B_Y if the function generated in bucket B_X by BE is placed in B_Y. Therefore, in a bucket tree, every vertex B_X other than the root has one parent vertex B_Y and possibly several child vertices $B_{Z_1}, ..., B_{Z_t}$.

The structure of the bucket tree can be extracted also from the induced-ordered graph of $\mathcal{M}$ along d using the following definition.

Definition 5.2 bucket-tree, separator, eliminator. Let $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \prod \rangle$ be a graphical model whose primal graph is G, and let $d = (X_1, ..., X_n)$ be an ordering of its variables. Let (G^*, d) be the induced graph along d of G.

- The bucket tree has the buckets denoted $\{B_i\}_{i=1}^n$ as its nodes. Each bucket contains a set of functions and a set of variables. The functions are those placed in the bucket according to the bucket partitioning rule where ψ_i is their product. The set of variables in B_i, denoted $scope(B_i)$, is X_i and all its parents in the induced-graph (G^*, d). Each vertex B_i points to B_j (or, B_j is the parent of B_i) if X_j is the closest neighbor of X_i that appear before it in (G^*, d).
- If B_j is the parent of B_i in the bucket tree, then the separator of X_i and X_j, $sep(B_i, B_j) = scope(B_i) \cap scope(B_j)$.
- Given a directed edge (B_i, B_j) in the bucket tree, $elim(i, j)$ is the set of variables in B_i and not in B_j, namely $elim(B_i, B_j) = scope(B_i) - sep(B_i, B_j)$. We will call this set "the eliminator from B_i to B_j."

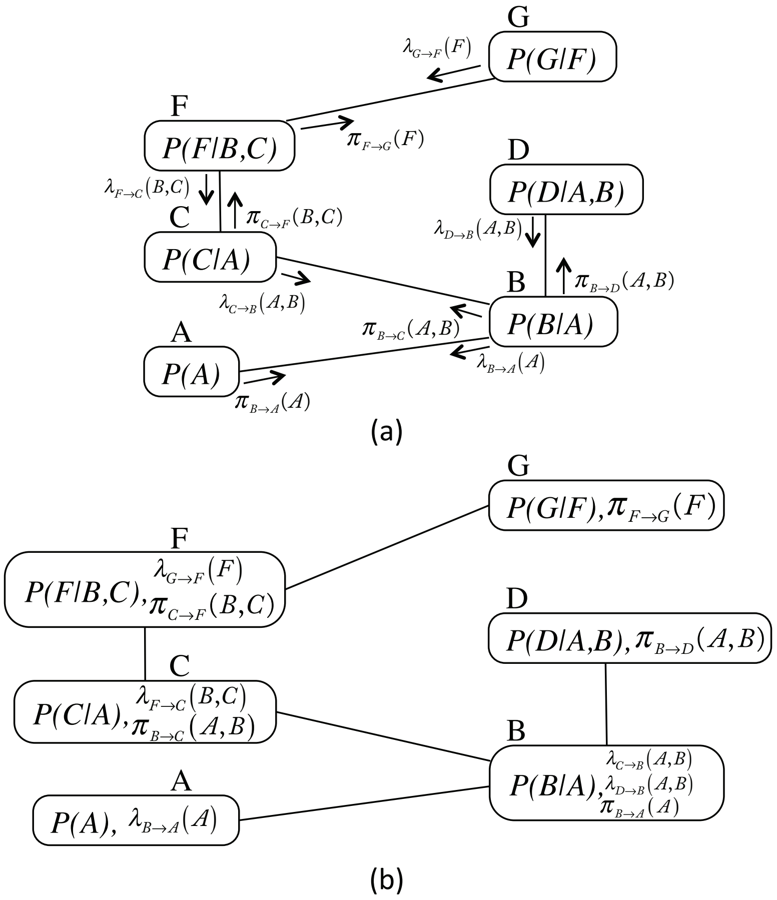


Figure 5.2: Propagation of π's and λ's along the bucket tree (a), and the augmented output bucket tree (b).

Algorithm *bucket-tree elimination (BTE)* presented in Figure 5.3 includes the two message passing phases along the bucket tree. Notice that we always assume that the input may contain also a set of evidence nodes because this is typical to a large class of problems in probabilistic networks. Given an ordering of the variables, the first step of the algorithm generates the bucket tree by partitioning the functions into buckets and connecting the buckets into a tree. The subsequent *top-down* phase is identical to general bucket elimination. The *bottom-up* messages are defined as follows. The messages sent from the root up to the leaves will be denoted by π. The message from B_j to a child B_i is generated by multiplying the bucket's function ψ_j by all the π messages from its parent bucket and all the λ messages from its *other* child buckets and marginalizing (e.g., summing) over the eliminator from B_j to B_i. We see that downward messages are generated by eliminating a single variable. Upward messages, on the other hand, may be generated by eliminating zero, one or more variables.

When BTE terminates, each output bucket $B^{'}_{i}$ contains the $\pi_{j \to i}$ it received from its parent B_j, its own function ψ_j and the $\lambda_{k \to i}$ messages sent from each child B_k. Then, each bucket can compute its belief over all the variables in its bucket, by multiplying all the functions in a bucket as specified in step 3 of BTE. It can then compute also the belief on single variables and the probability of evidence as in the procedure in Figure 5.4.

```
ALGORITHM BUCKET-TREE ELIMINATION (BTE)
Input: A problem $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \prod, \sum \rangle$, ordering $d$.
$X = \{X_1, ..., X_n\}$ and $F = \{f_1, ..., f_r\}$
Evidence $E = e$.
Output: Augmented buckets $\{B'_i\}$, containing the original functions and all the
$\pi$ and $\lambda$ functions received from neighbors in the bucket tree.
1. Pre-processing: Partition functions to the ordered buckets as usual
   and generate the bucket tree.
2. Top-down phase: $\lambda$ messages (BE) do
       for $i = n$ to 1, in reverse order of $d$ process bucket $B_i$:
           The message $\lambda_{i \to j}$ from $B_i$ to its parent $B_j$, is:
           $\lambda_{i \to j} \Leftarrow \sum_{elim(i,j)} \psi_i \cdot \prod_{k \in child(i)} \lambda_{k \to i}$
       endfor
3. bottom-up phase: $\pi$ messages
       for $j = 1$ to $n$, process bucket $B_j$ do:
           $B_j$ takes $\pi_{k \to j}$ received from its parent $B_k$, and computes a message $\pi_{j \to i}$
           for each child bucket $B_i$ by
           $\pi_{j \to i} \Leftarrow \sum_{elim(j,i)} \pi_{k \to j} \cdot \psi_j \cdot \prod_{r \neq i} \lambda_{r \to j}$
       endfor
4. Output: augmented buckets $B'_1, ..., B'_n$, where each $B'_i$ contains the
   original bucket functions and the $\lambda$ and $\pi$ messages it received.
```

Figure 5.3: Algorithm bucket-tree elimination.

Example 5.3 Figure 5.2a shows the complete execution of BTE along the bucket tree. Notice, that the variables in each bucket are not stated explicitly in the figure. For example, the variables in the $bucket_C$ are A, B, C while the scope of the original function has only variables A, C. The π and λ messages are placed on the outgoing upward arcs. The π functions in the bottom-up phase

```
COMPUTING MARGINAL BELIEFS
Input: a bucket tree processed by BTE with augmented buckets: $B\prime_1, ..., B\prime_n$
output: beliefs of each variable, bucket, and probability of evidence.
        $bel(B'_i) \Leftarrow \alpha \prod_{f \in B'_i} f$
        $bel(X_i) \Leftarrow \alpha \sum_{B'_i - \{X_i\}} \prod_{f \in B'_i} f$
        $P(evidence) \Leftarrow \sum_{B'_i} \prod_{f \in B'_i} f$
```

Figure 5.4: Query answering.

are computed as follows (the first three were demonstrated earlier):
$\pi_{A \to B}(a) = P(a)$
$\pi_{B \to C}(c, a) = P(b|a)\lambda_{D \to B}(a, b)\pi_{A \to B}(a)$
$\pi_{B \to D}(a, b) = P(b|a)\lambda_{C \to B}(a, b)\pi_{A \to B}(a, b)$
$\pi_{C \to F}(c, b) = \sum_a P(c|a)\pi_{B \to C}(a, b)$
$\pi_{F \to G}(f) = \sum_{b,c} P(f|b, c)\pi_{C \to F}(c, b)$

The actual output (the augmented buckets) are shown in Figure 5.2b.

Explicit submodels. Extending the view of the above algorithms for any reasoning task over graphical models, we can show that when BTE terminates we have in each bucket all the information needed to answer any reasoning task on the variables appearing in that bucket. In particular, we do not need to look outside a bucket to answer a belief query. We call this property "explicitness". It is sometime referred to also as *minimality* or *decomposability* [Montanari, 1974].

Definition 5.4 Explicit function and explicit sub-model. Given a graphical model $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \prod \rangle$, and reasoning tasks defined by marginalization $\sum$ and given a subset of variables Y, $Y \subseteq \mathbf{X}$, we define $\mathcal{M}_Y$, the explicit function of $\mathcal{M}$ over Y:

$$\mathcal{M}_Y = \sum_{X-Y} \prod_{f \in F} f, \tag{5.4}$$

We denote by F_Y any set of functions whose scopes are subsumed in Y over the same domains and ranges as the functions in $\mathbf{F}$. We say that (Y, F_Y) is an explicit submodel of $\mathcal{M}$ iff

$$\prod_{f \in F_Y} f = \mathcal{M}_Y \tag{5.5}$$

As we elaborate more later, once we have an explicit representation in each cluster we can answer most queries locally. We can compute the belief of each variable, the probability of evidence or the partition function, and the optimal solution costs inside each bucket.

Theorem 5.5 Completeness of BTE. *Given $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \prod, \sum \rangle$ and evidence $E = e$, when algorithm BTE terminates, each output bucket is explicit relative to its variables. Namely, for each B_i, $\Pi_{f \in B'_i} f = \mathcal{M}_{scope(B_i)}$.*

The completeness of BTE will be derived from that of a larger class of tree propagation algorithms that we present in the following section. We next address the complexity of BTE.

Theorem 5.6 Complexity of BTE. *Let $w^*(d)$ be the induced width of (G^*, d) where G is the primal graph of $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \prod, \sum \rangle$, r be the number of functions in $\mathbf{F}$ and k be the maximum domain size. The time complexity of BTE is $O(r \cdot deg \cdot k^{w^*(d)+1})$, where deg is the maximum degree of a node in the bucket tree. The space complexity of BTE is $O(n \cdot k^{w^*(d)})$.*

Proof: As we previously showed, the downward λ messages take $O(r \cdot k^{w^*(d)+1})$ steps. This simply is the complexity of BE. The upward messages per bucket are computed for each of its child nodes. Each such message takes $O(r_i \cdot k^{w^*(d)+1})$ steps, where r_i is the number of functions in bucket B_i, yielding a time complexity per upward bucket of $O(r_i \cdot deg \cdot k^{w*d)+1})$. Summing over all buckets we get complexity of $O(r \cdot deg \cdot k^{w^*(d)+1})$. Since the size of each downward message is k^{w^*} we get space complexity of $O(n \cdot k^{w^*(d)})$. The complexity of BTE can be improved to be $O(rk^{w^*(d)+1})$ time and $O(nk^{w^*(d)+1})$ space [Kask at. al., 2005]. $\square$.

In theory the speedup expected from running BTE vs. running BE n times is at most n. This may seem insignificant compared with the exponential complexity in w^*, however it can be very significant in practice, especially when n is large. Beyond the saving in computation, the bucket tree provides an architecture for distributed computation of the algorithm when each bucket is implemented by a different cpu.

5.1.1 ASYNCHRONOUS BUCKET-TREE PROPAGATION

Algorithm *BTE* can also be described without committing to a particular schedule by viewing the bucket tree as an undirected tree and by unifying the up and down messages into a single message-type denoted by λ. In this case each bucket receives a λ message from each of its neighbors and each sends a λ message to every neighbor. This distributed algorithm, called Bucket-Tree Propagation or *BTP*, is written for a single bucket described in Figure 5.5. It is easy to see that the algorithm accomplishes the same as BTE and is therefore correct and complete. It sends at most two messages on each edge in the tree (computation starts from the leaves of the tree). It is also easy to see the distributed nature of the algorithm.

Theorem 5.7 Completeness of BTP. *Algorithm BTP terminates generating explicit buckets.*

Proof. The proof of BTP correctness follows from the correctness of BTE. All we need to show is that at termination the buckets' content in BTE and BTP are the same. (Prove as an exercise.) □

```
BUCKET-TREE PROPAGATION (BTP)
Input: A problem $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \prod, \sum \rangle$. $X = \{X_1, ..., X_n\}$ and
    $F = \{f_1, ..., f_r\}$, $\mathbf{E} = \mathbf{e}$. An ordering $d$ and a corresponding bucket-tree structure,
    in which for each node $X_i$, its bucket $B_i$ and its neighboring buckets are well defined.
Output: Explicit buckets. Assume functions assigned with the evidence.
1.  for bucket $B_i$ do:
2.      for each neighbor bucket $B_j$ do,
            once all messages from all other neighbors were received, do
                compute and send to $B_j$ the message
                $\lambda_{i \to j} \Leftarrow \sum_{elim(i,j)} \psi_i \cdot (\prod_{k \neq j} \lambda_{k \to i})$
3.  Output: augmented buckets $B'_1, ..., B'_n$, where each $B'_i$ contains the
    original bucket functions and the $\lambda$ messages it received.
```

Figure 5.5: Algorithm Bucket-tree propagation (BTP).

Finally, since graphical models whose primal graph is a tree, the induced-width is 1, clearly

Proposition 5.8 BTE on trees *For graphical models whose primal graph is a tree, algorithms BTE and BTP are time and space $O(nk^2)$ and $O(nk)$, respectively, when k bound the domain size and n bounds the number of variables.*

5.2 FROM BUCKET TREES TO CLUSTER TREES

Algorithms BTE and its asynchronous version BTP are special cases of a class of algorithms that operates over a tree-decomposition of the graphical model. A tree-decomposition takes a graphical model and embeds it in a tree where each node in the tree is a cluster of variables and functions. The decomposition allows message-passing between the clusters in a manner similar to BTE and BTP, yielding an algorithm which we will call Cluster-Tree Elimination or CTE. We provide the idea through a short route from BTE to CTE and subsequently establish the formal grounds in more details.

5.2.1 FROM BUCKETS TO CLUSTERS; THE SHORT ROUTE

Example 5.9 Let's go back to example 5.1. We saw that to facilitate message-passing in the reverse order of d we suggested first to virtually collapse the individual buckets of D, A, B into a single cluster and reasoned from there. However, we can actually apply this collapsing and then perform the computation defined in Equations 5.1 and 5.2 within the collapsed cluster, instead of moving around information in between the individual buckets. This will yield 4 clusters. The first 3 correspond to the original buckets of variables G, F and C, and the forth is the cluster over $\{A, D, B\}$ groups all the functions in the respective buckets. The message sent from the new cluster ABD to $bucket_C$ is identical to the π message from $bucket_B$ to $bucket_C$, and it can be computed using the the same rule as specified in algorithm BTP. Namely, the message from ABD to C is computed by taking the product of all the functions in cluster ABD and then eliminating the eliminator (namely, summing over D). We get,

$$\pi_{ABD \rightarrow C} = \sum_{D} P(b|a) \cdot P(d|a, b) \cdot P(a).$$

It is easy to see that this is the same $\pi_{B \rightarrow C}$ computed by BTP.

In other words, BTE/BTP algorithms can be extended to work over a larger ensemble of cluster-trees and those can be obtained, by just collapsing some adjacent clusters into larger clusters, where the starting point is a bucket-tree. In certain cases, as in the example above, there is no loss in efficiency. In fact we have less clusters yielding a simplification. In other cases, if the clusters get larger and larger, we may loose decomposability and therefore have less effective computation. We will identify the tradeoffs associated with this process and provide a scheme to generate good cluster-tree decompositions. What we wish to stress here is that the only thing that will change is the complexity of the resulting algorithms.

In summary, the algorithm CTE that we will present, is just the BTP that is applied to any cluster-tree and those can be obtained by collapsing adjacent buckets (their functions and variables) of a bucket-trees. A preliminary version of the algorithm, called *CTP* is presented in Figure 5.6.

Clearly,

Theorem 5.10 Complexity of CTP. *Algorithm CTP terminates generating explicit clusters. Its time complexity is exponential in w, which is the maximum number of variables in a cluster.*

In the rest of this chapter we provide a somewhat different route to CTE and to general message-passing over general tree-decompositions, whose starting point are acyclic graphical models.

Some graphical models are inherently tree-structured. Namely, their input functions already has a dependency structure that can be captured by a tree-like graph (with no cycles). Such graphical models are called *Acyclic Graphical models* [Maier, 1983] and they include regular trees as a special case. We will describe acyclic models first and then show how a tree-decomposition can impose a

```
CLUSTER-TREE PROPAGATION (CTP)
Input: A problem $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \prod, \sum \rangle$, an ordering $d$. $X = \{X_1, ..., X_n\}$ and
$F = \{f_1, ..., f_r\}$, $\mathbf{E} = \mathbf{e}$.
    A cluster-tree structure, $C_1, ...C_l$ obtained from the bucket-tree structure along ordering $d$,
    by collapsing some adjacent buckets into clusters.
Output: Explicit clusters.
1.  for bucket $C_i$ do:
2.      for each neighbor bucket $C_j$ do,
            once all messages from all other neighboring clusters were received, do
                compute and send to $C_j$ the message
                $\lambda_{i \to j} \Leftarrow \sum_{elim(i,j)} \psi_i \cdot (\prod_{k \neq j} \lambda_{k \to i})$
3.  Output: augmented clusters $C'_1, ..., C'_n$, where each $C'_i$ contains the
    original cluster functions and the $\lambda$ messages it received.
```

Figure 5.6: Algorithm Cluster-Tree propagation (CTP).

tree structure on non-acyclic graphical models as well. In particular we will show that bucket-trees are special cases of tree-decompositions.

5.2.2 ACYCLIC GRAPHICAL MODELS

As we know, a graphical model can be associated with a dual graph (see Definition 2.7). which provides an alternative view of the graphical model. In this view, each function resides in its own node which can be regarded as a meta variable and the arcs indicate equality constraints between shared variables. So, if a graphical model's dual graph is a tree, we have a tree-graphical model, and we know it can be solved in linear time by a BTE-like message-passing algorithm over the dual graph (see Proposition 5.8.)

Sometime the dual graph seems to not be a tree, but it is in fact, a tree. This is because some of its arcs are redundant and can be removed while not violating the original independency relationships that is captured by the graph. Arcs are redundant if they express a dependency between two nodes that is already captured or implied by an alternative set of dependencies, and therefore their removal does not alter the conditional independence captured by graph separation (for more see [Pearl, 1988]). We illustrate with the next example and then provide the formal definition capturing this notion.

Example 5.11 Refer to Figure 5.7 (presented earlier as Figure 2.1). We see that the arc between (AEF) and (ABC) in Figure 2.1c expresses redundant dependency because variable A also appears along the alternative path $(ABC) - AC - (ACE) - AE - (AEF)$. In other words, a dependency

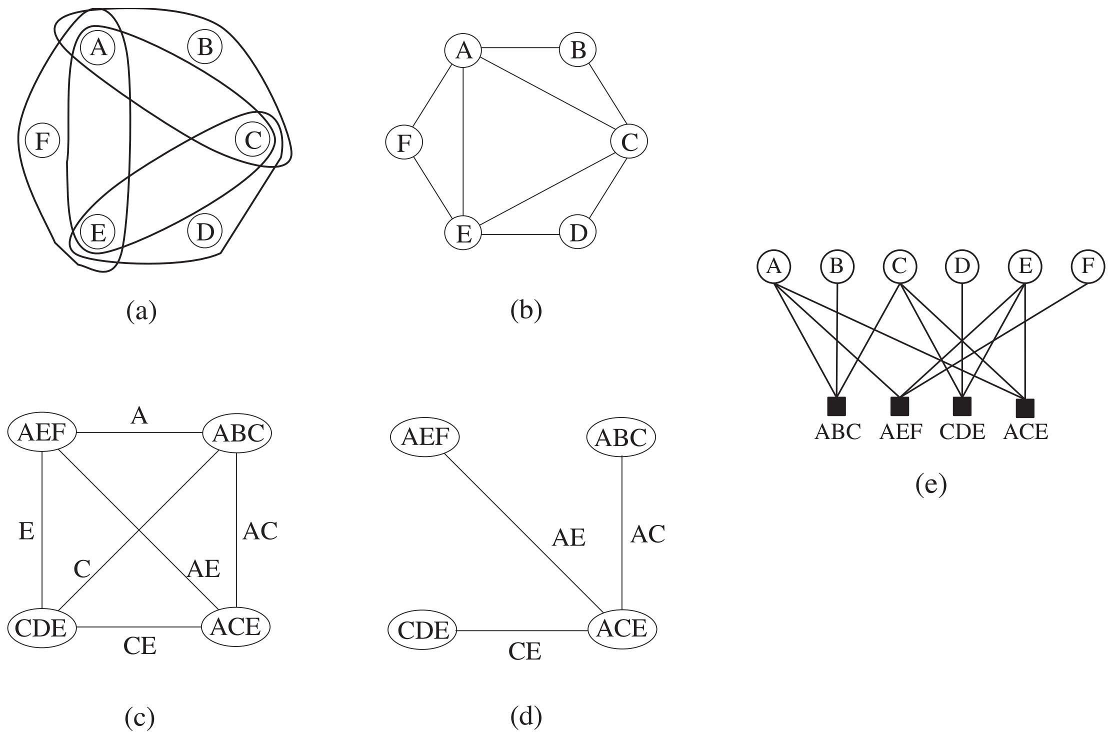


Figure 5.7: (a) Hyper; (b) primal; (c) dual; (d) join-tree of a graphical model having scopes ABC, AEF, CDE and ACE; and (e) the factor graph.

between AEF and ABC relative to A is maintained through the path even if they are not directly connected. Likewise, the arcs labeled E and C are also redundant. Their removal yields the tree in 2.1d which we call a join-tree. This reduced dual graph in 2.1d satisfies a property called connectedness.

Definition 5.12 Connectedness, join-trees. Given a dual graph of a graphical model $\mathcal{M}$, an arc subgraph of the dual graph satisfies the *connectedness* property iff for each two nodes that share a variable, there is at least one path of labeled arcs of the dual graph such that each contains the shared variables. An arc subgraph of the dual graph that satisfies the connectedness property is called a *join-graph* and if it is a tree, it is called a *join-tree*.

We can now formally define acyclic graphical models.

Definition 5.13 Acyclic networks. A graphical model , $\mathcal{M} = \langle X, D, F, \prod \rangle$, whose dual graph *has* a join-tree is called an *acyclic graphical model*.

Example 5.14 We can see that the join-tree in Figure 5.7d satisfies the connectedness property. The graphical model defined by the scopes: AEF, ABC, CDE, ACE is therefore, acyclic.

It is easy to see that if algorithm BTE is applied to an acyclic graphical model it would be efficient. In fact, it will be time and space linear. This is because the join-tree suggests an ordering over the variables where the width of the primal graph along d equals the induced-width (prove as an exercise) and where each function is placed in a single bucket. This means that messages are always defined over scopes that are subsumed by existing scopes of the original functions.

Theorem 5.15 *Given an acyclic graphical model, algorithm BTE is time and space linear. (For an informal proof see the Appendix.)*

5.2.3 TREE DECOMPOSITION AND CLUSTER TREE ELIMINATION

Now that we have established that acyclic graphical models can be solved efficiently, all that remains is to transform a general graphical model into an acyclic one. This task is facilitated by the concept of tree-decomposition defined next. We will subsequently show that a bucket tree is a special case of tree decomposition.

Intuitively a tree-decomposition is a tree such that each function belongs to a single node and its scope is a subset of a set of variables associated with the node.

Definition 5.16 Tree decomposition, cluster tree. Let $\mathcal{M} =< X, D, F, \prod >$ be a graphical model. A *tree-decomposition* of $\mathcal{M}$ is a triple $< T, \chi, \psi >$, where $T = (V, E)$ is a tree, and χ and ψ are labeling functions which associate with each vertex $v \in V$ two sets, $\chi(v) \subseteq X$ and $\psi(v) \subseteq F$ satisfying:

1. Each function belongs to a single node and its scope is a subset of the variables of the node it belongs to. Formally, for each function $f_i \in F$, there is *exactly* one vertex $v \in V$ such that $f_i \in \psi(v)$, and $scope(f_i) \subseteq \chi(v)$; and
2. for each variable $X_i \in X$, the set $\{v \in V | X_i \in \chi(v)\}$ induces a connected subtree of T. This is also called the running intersection property [Maier, 1983].

We will often refer to a node and its functions as a *cluster* and use the term *tree decomposition* and *cluster tree* interchangeably.

Definition 5.17 treewidth, pathwidth, separator-width, eliminator. The *treewidth* [Arnborg, 1985] of a tree decomposition $< T, \chi, \psi >$ is $max_{v \in V} |\chi(v)|$ minus 1. Given two adjacent vertices u and v of a tree-decomposition, the *separator* of u and v is $sep(u, v) = \chi(u) \cap \chi(v) = sep(v, u)$, and the *eliminator* of u with respect to v is $elim(u, v) = \chi(u) - \chi(v)$. The separator-width is the maximum over all separators. The pathwidth of a graph is the treewidth when only chain-line tree-decompositions are restricted.

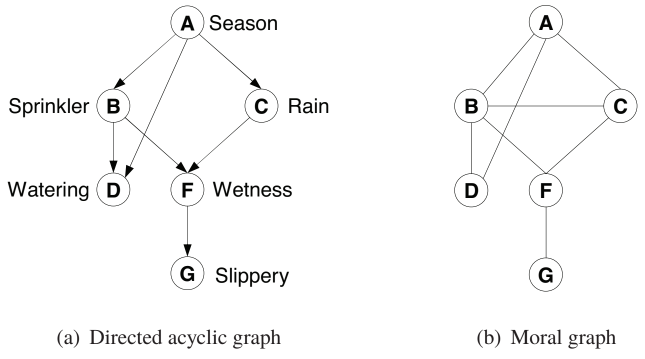


Figure 5.8: Belief network $P(G, F, C, B, A) = P(G|F)P(F|C, B)P(D|A, B)P(C|A)P(B|A)P(A)$.

Example 5.18 Consider again the Bayesian network in Figure 5.8a. Any of the cluster-trees in Figure 5.9 describes a partition of variables into clusters. We can now place each input function into a cluster that contains its scopes, and verify that each is a legitimate tree decomposition. For example, Figure 5.9c shows a cluster-tree decomposition with two vertices, and labeling $\chi(1) = \{G, F\}$ and $\chi(2) = \{A, B, C, D, F\}$. Any function with scope $\{G\}$ must be placed in vertex 1 because vertex 1 is the only vertex that contains variable G (placing a function having G in its scope in another vertex will force us to add variable G to that vertex as well). Any function with scope $\{A, B, C, D\}$ or one of its subsets must be placed in vertex 2, and any function with scope $\{F\}$ can be placed either in vertex 1 or 2. Notice that the tree-decomposition at Figure 5.9a is actually a bucket-tree. Note that the separator between any two nodes is labeled on the edge connecting them.

We see that for some nodes $sep(u, v) = \chi(u)$. That is, all the variables in vertex u belong to an adjacent vertex v. In this case the number of clusters in the tree decomposition can be reduced by absorbing vertex u into v without increasing the cluster size in the tree-decomposition. This is accomplished by moving from Figure 5.9a to Figure 5.9b, when the nodes whose scopes are $\{A\}$ and $\{A, B\}$ are absorbed into the node of $\{A, B, C\}$.

Definition 5.19 Minimal tree decomposition. A tree decomposition is *minimal* if $sep(u, v) \subsetneqq \chi(u)$ and $sep(u, v) \subsetneqq \chi(v)$ for each pair (u, v) of adjacent nodes.

We can show the following.

Theorem 5.20 *A bucket tree of a graphical model $\mathcal{M}$, is a tree decomposition of $\mathcal{M}$. (for a proof see Appendix.)*

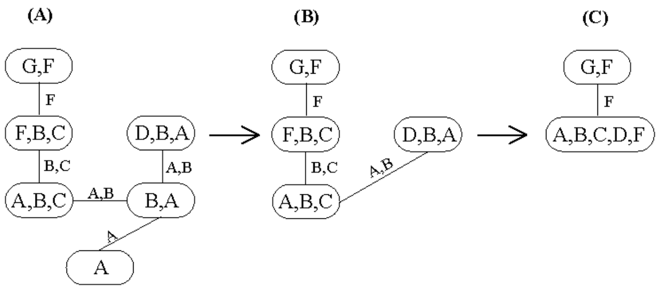


Figure 5.9: From a bucket tree to join tree to a super bucket tree.

```
CLUSTER-TREE ELIMINATION (CTE)
Input: A tree decomposition $< T, \chi, \psi >$ for a problem $M =< X, D, F, \prod, \sum\} >$,
$X = \{X_1, ..., X_n\}$, $F = \{f_1, ..., f_r\}$. Evidence $E = e$, $\psi_u = \prod_{f \in \psi(u)} f$
Output: An augmented tree decomposition whose clusters are all model explicit.
Namely, a decomposition $< T, \chi, \bar{\psi} >$ where $u \in T$, $\bar{\psi}(u)$ is model explicit relative to $\chi(u)$.
1. Initialize. (denote by $m_{u \to v}$ the message sent from vertex $u$ to vertex $v$.)
2. Compute messages:
        For every node $u$ in $T$, once $u$ received messages from all neighbors but $v$,
            Process observed variables:
            For each node $u \in T$ assign relevant evidence to $\psi(u)$
            Compute the message:
                $m_{u \to v} \leftarrow \sum_{\chi(u) - sep(u,v)} \psi_u \cdot \prod_{r \in neighbor(u), r \neq v} m_{r \to u}$
        endfor
   Note: functions whose scopes do not contain any separator variable
   do not need to be combined and can be directly passed on to the receiving vertex.
3. Return: The explicit tree $< T, \chi, \bar{\psi} >$, where
   $\bar{\psi}(v) \Leftarrow \psi(v) \cup_{u \in neighbor(v)} \{m_{u \to v}\}$
   return the explicit function: for each $v$, $M_{\chi(v)} = \prod_{f \in \bar{\psi}(v)} f$
```

Figure 5.10: Algorithm Cluster-Tree Elimination (CTE).

CTE. We will show now that a tree decomposition facilitates a message-passing scheme, called *Cluster-Tree Elimination* (CTE), that is similar to BTE in the same sense the CTP is similar to BTP. The algorithm is presented in Figure 5.10. Like in BTP, each vertex of the tree sends a function to each of its neighbors. All the functions in a vertex u and all the messages received by u from all its neighbors other than a specific vertex v to which u's message is directed, are combined by product. The combined function is marginalized over the separator of vertices u and v (namely, eliminating the eliminator) using the marginalization operator, $\sum$, and the marginalized function is then sent from u to v. We will denote messages by m here.

Vertex activation can be asynchronous and convergence is guaranteed. If processing is performed from leaves to root and back, convergence is guaranteed after two passes, where only one message is sent on each edge in each direction. If the tree contains l edges, then a total of $2l$ messages will be sent.

Example 5.21 Consider again the graphical model whose primal graph appears in Figure 5.8(b) but now assume that all functions are defined on pairs of variables (you can think of this as a Markov network). Two tree decompositions are given in Figure 5.11a and 5.11b. For the tree-decomposition in 5.11b we show the propagated messages explicitly in Figure 5.11c. Since cluster 1 contains only one function, the message from cluster 1 to 2 is the summation of f_{FG} over the separator between cluster 1 and 2, which is $\{F\}$. The message $m_{2\to 3}$ from cluster 2 to cluster 3 is generated by combining the functions in cluster 2 with the message $m_{1\to 2}$, and then marginalizing over the separator between cluster 2 and 3, which is $\{B, C\}$, and so on.

Once all vertices have received messages from all their neighbors we have the explicit clusters (see Definition 5.4) and therefore an answer to any singleton marginal query (e.g., beliefs) and a host of other reasoning tasks can be accomplished over the explicit tree in linear time. Before we prove these properties in Section 5.3 we will pause to discuss the generation of tree-decompositions.

5.2.4 GENERATING TREE DECOMPOSITIONS

We have already established that a bucket-tree built along a given ordering is a tree decomposition. Each node is a bucket, whose variables include the bucket's variable and all its earlier neighbors in the induced graph and whose functions are those assigned to it by the initial bucket-partitioning. As suggested earlier, once we have a tree decomposition, other tree decompositions can be obtained by merging adjacent nodes. Therefore, bucket trees can serve as a starting point for generating arbitrary tree decompositions, a process that is justified by the following proposition.

Proposition 5.22 *If T is a tree decomposition, then any tree obtained by merging adjacent clusters is also a tree decomposition. (Prove as an exercise.)*

A special class of tree-decompositions called *join-trees* can be obtained by merging subsumed buckets into their containing buckets, as defined by their variables. This is indeed the approach we discussed in our short route. Alternatively, they can be generated directly from the induced-graph by selecting as clusters only those buckets that are associated with maximal cliques.

Algorithm join-tree clustering (JTC) for generating join-tree decompositions, is described in Figure 5.12. The algorithm generates an induced graph, which we know is chordal, it then identifies its maximal cliques as the candidate cluster-nodes and then connect them in a tree structure. This

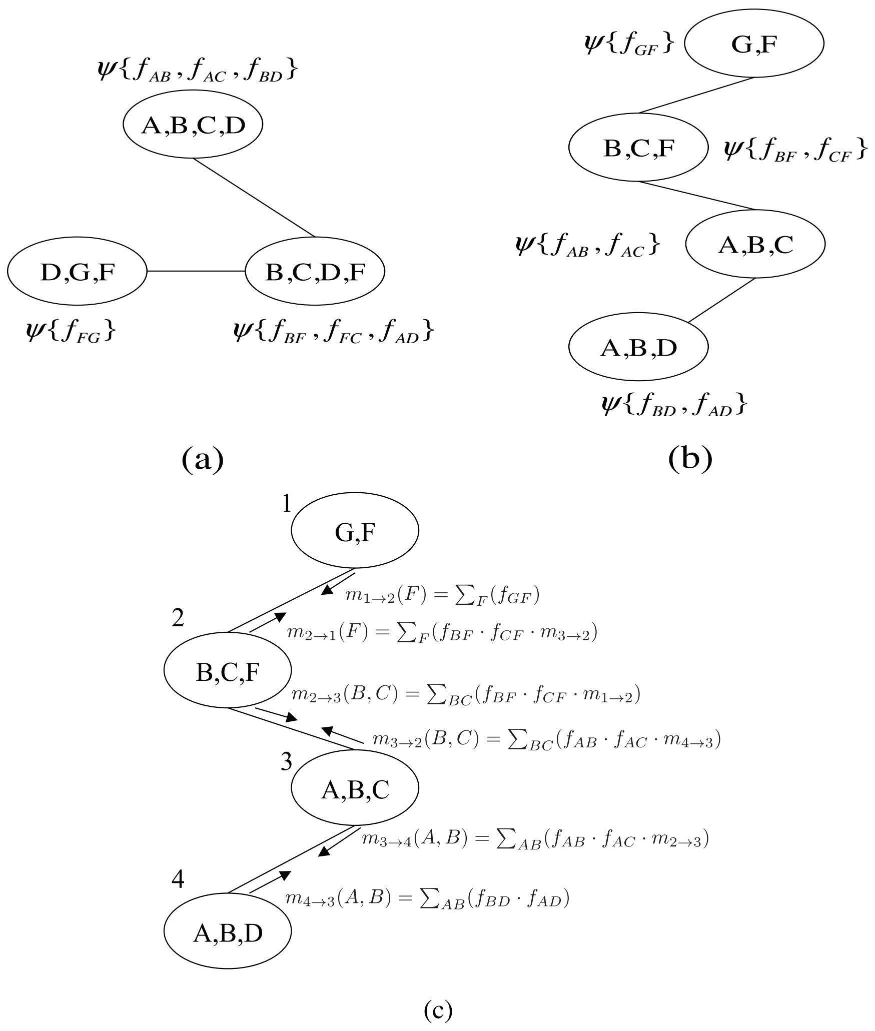

$\psi\{f_{AB}, f_{AC}, f_{BD}\}$
A,B,C,D
D,G,F
B,C,D,F
$\psi\{f_{FG}\}$
$\psi\{f_{BF}, f_{FC}, f_{AD}\}$
(a)
$\psi\{f_{GF}\}$
G,F
B,C,F
$\psi\{f_{BF}, f_{CF}\}$
$\psi\{f_{AB}, f_{AC}\}$
A,B,C
A,B,D
$\psi\{f_{BD}, f_{AD}\}$
(b)
1
G,F
$m_{1\to2}(F) = \sum_F(f_{GF})$
2
$m_{2\to1}(F) = \sum_F(f_{BF} \cdot f_{CF} \cdot m_{3\to2})$
B,C,F
$m_{2\to3}(B,C) = \sum_{BC}(f_{BF} \cdot f_{CF} \cdot m_{1\to2})$
$m_{3\to2}(B,C) = \sum_{BC}(f_{AB} \cdot f_{AC} \cdot m_{4\to3})$
3
A,B,C
$m_{3\to4}(A,B) = \sum_{AB}(f_{AB} \cdot f_{AC} \cdot m_{2\to3})$
4
$m_{4\to3}(A,B) = \sum_{AB}(f_{BD} \cdot f_{AD})$
A,B,D
(c)

Figure 5.11: Two tree decompositions of a graphical model.

process determines the variables associated with each node. Subsequently, functions are partitioned into the clusters appropriately.

Example 5.23 Consider the graph in Figure 5.13a and the ordering $d_1 = (G, D, F, C, B, A)$ in Figure 5.13b. Performing the triangulation step of JTC connects parents recursively from the last variable to the first, creating the induced-ordered graph by adding the new (broken) edges of Figure 5.13b. The maximal cliques of this induced graph are: $Q_1 = \{A, B, C, D\}$, $Q_2 = \{B, C, F, D\}$ and $Q_3 = \{F, D, G\}$. Alternatively, if ordering $d_2 = (A, B, C, F, D, G)$ in Figure 5.13c is used,

```
JOIN-TREE CLUSTERING (JTC)
Input: A graphical model $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \prod \rangle$, $\mathbf{X} = \{X_1, ..., X_n\}$, $\mathbf{F} = \{f_1, ..., f_r\}$.
Its scopes $S = S_1, ..., S_r$ and its primal graph is $G = (X, E)$.
Output: A join-tree decomposition $< T, \chi, \psi >$ for $\mathcal{M}$
1. Select a variable ordering, $d = (X_1, ..., X_n)$.
2. Triangulation (create the induced graph along $d$ and call it $G^*$):
        for $j = n$ to 1 by -1 do
        $E \leftarrow E \cup \{(i,k)|i < j,\ k < j,\ (i,j) \in E, (k,j) \in E\}$
3. Create a join-tree of the induced graph $(G^*, d)$ as follows:
        a. Identify all maximal cliques in the chordal graph.
        Let $C = \{C_1, ..., C_t\}$ be all such cliques.
        b. Create a tree $T$ of cliques as follows:
        Connect each $C_i$ to a $C_j$ $(j < i)$ with whom it shares largest subset of variables.
4. Create $\psi_i$: Place each input function into a cluster-node whose variables contain its scope.
5. Return a tree-decomposition $< T, \chi, \psi >$, where $T$ is generated in step 3,
$\chi(i) = C_i$ and $\psi(i)$ is determined in step 4.
```

Figure 5.12: Join-tree clustering.

the induced graph generated has only one added edge. The cliques in this case are: $Q_1 = \{G, F\}$, $Q_2 = \{A, B, D\}$, $Q_3 = \{B, C, F\}$ and $Q_4 = \{A, B, C\}$. Yet, another example is given in Figure 5.13d. The corresponding join-trees of orderings d_1 and d_2 are depicted in the earlier decompositions observed in Figure 5.11a and 5.11b, respectively.

Treewidth and induced-width. Clearly, if the ordering used by JTC has induced width $w(d)$, the treewidth (Definition 5.17) of the resulting join-tree is $w(d)$ as well. And, vice-versa, given a tree-decomposition of a graph whose treewidth is w, there exist an ordering of the nodes d whose induced-width satisfies $w(d) = w$. In other words *treewidth and induced-width can be viewed as synonym concepts for graphs*, yet induced-width is explicitly defined for an ordering.

5.3 PROPERTIES OF CTE FOR GENERAL MODELS

Algorithm CTE which takes as an input a tree-decomposition of a graphical model, and evidence, outputs an *explicit* tree-decomposition. In this section we will prove this claim and discuss issues of complexity.

Convergence. Intuitively, it is clear that CTE converges after 2 message passing along the tree. If we remove the edge (u, v) we get 2 subtrees. One rooted at node u and one rooted at node v. The

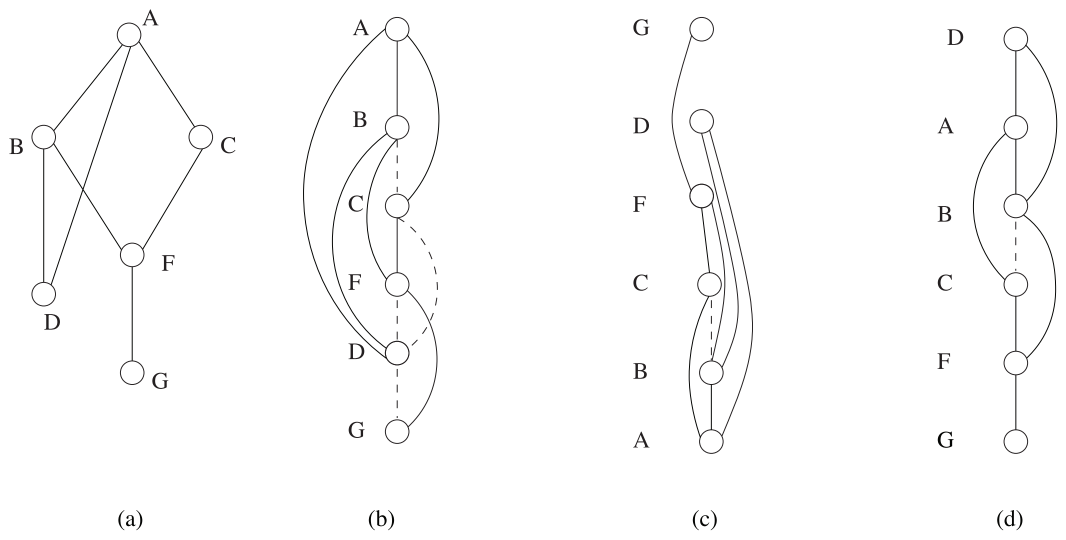


Figure 5.13: A graph (a) and three of its induced graphs (b), (c), and (d).

outgoing message from u to v depends solely on information in the subtree that include u but not v. Therefore the message $m_{u \to v}$ will not be influenced by the message $m_{v \to v}$.

Theorem 5.24 Convergence of CTE. *Algorithm CTE converges after two iterations.*

5.3.1 CORRECTNESS OF CTE

The correctness of CTE can be shown in two steps. First showing that CTE can solve acyclic graphical models. Since a tree-decomposition transforms a graphical model into an acyclic one, the correctness argument follows. However, instead of taking this route, we will next state the correctness based on the general properties of the *combine* $\bigotimes$ and *marginalize* $\Downarrow$ operators in order to emphasize the broad applicability of this algorithm. The following theorem articulates the properties which are required for correctness. The proof can be found in [Kask at. al., 2005] and in the Appendix of this chapter.

Theorem 5.25 Soundness and completeness. *Given a graphical model and reasoning tasks $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \bigotimes, \Downarrow \rangle$, and assuming that the combination operator $\bigotimes$ and the marginalization operator $\Downarrow_Y$ satisfy the following properties [Shenoy, 1997]):*

1. *order of marginalization does not matter:*
   $\Downarrow_{X-\{X_i\}} (\Downarrow_{X-\{X_j\}} f(X)) = \Downarrow_{X-\{X_j\}} (\Downarrow_{X-\{X_i\}} f(X))$*;*
2. *commutativity:* $f \bigotimes g = g \bigotimes f$*;*

3. *associativity:* $f \bigotimes (g \bigotimes h) = (f \bigotimes g) \bigotimes h$;
4. *restricted distributivity:*
$\Downarrow_{X-\{X_k\}} [f(X - \{X_k\}) \bigotimes g(X)] = f(X - \{X_k\}) \bigotimes \Downarrow_{X-\{X_k\}} g(X)$.

Algorithm CTE is sound and complete. Namely, it is guaranteed to transform a tree-decomposition $< T, \chi, \psi >$ into an explicit one. Namely, for every node $v \in T$, given the messages generated $m_{u \to v}$ for all $(u, v) \in T$, then

$$M_{\chi(u)} = \psi_u \otimes (\bigotimes_{\{j | (j,u) \in E\}} m_{j \to u}) \ .$$

where $M_{\chi(u)}$ is explicit relative to $\chi(u)$ (Definition 5.4). For a proof see the Appendix. □

It is common to call the combined function in a cluster as *belief*. Another concept associated with clusters is their normalized constant.

Definition 5.26 belief, normalizing constants. Given a tree decomposition $< T, \chi, \Psi >, T = (V, E)$ of a graphical model $\mathcal{M} =< X, D, F, \bigotimes, \Downarrow >$, and a set of messages denoted by m (potentially generated by CTE, but not only) then the beliefs associated with each cluster $u \in T$, relative to incoming messages $\{m\}$ is:

$$b_u = \psi_u \otimes [\bigotimes_{k \in neighbors(v)} m_{k \to u}] \tag{5.6}$$

We also define

$$b_{sep(u,v)} = \Downarrow_{\chi(u)-sep(u,v)} b_u. \tag{5.7}$$

The normalizing constant of a node v is defined by:

$$K_u = \Downarrow_{\chi(u)} b_u \tag{5.8}$$

What we showed is that if the set of messages m were generated by CTE then the beliefs of each node is the explicit function of the model. Clearly, therefore for any two adjacent nodes in the tree decomposition, marginalizing the beliefs over the separators must yields an identical function which is the explicit function over that separator. In the case of probabilities (when $\bigotimes$ is a product and $\Downarrow$ is summation) this means that the marginal probability on the separator variables can be obtained in either one of the clusters.

Definition 5.27 Pairwise consistency. Given a tree decomposition $< T, \chi, \psi >, T = (V, E)$ of $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \bigotimes, \Downarrow \rangle$, and a set of messages $m_{u \to v}, m_{v \to u}$, for every edge $(u, v) \in T$, then

$$\Downarrow_{sep(u,v)} [\psi_v \otimes (\bigotimes_{\{j | (j,v) \in E\}} m_{j \to v})] = \Downarrow_{sep(u,v)} [\psi_u \otimes (\bigotimes_{\{j | (j,u) \in E\}} m_{j \to u})].$$

using the definition of beliefs this is equivalent to:

$$\Downarrow_{sep(u,v)} b_v = b_{uv} = \Downarrow_{sep(u,v)} b_u.$$

Interestingly, we can prove pairwise consistency of CTE without using explicitness, as well as several properties and in particular that the normalizing constants of all nodes are identical.

Theorem 5.28 *Given a tree decomposition* $< T, \chi, \psi >$, $T = (V, E)$ *of* $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \bigotimes, \Downarrow \rangle$, *when* CTE *terminates with the set of messages* $m_{u \to v}, m_{v \to u}$, *then for any edge in* T,

1. *The message obey symmetry, namely:*
$$b_{sep(u,v)} = m_{u \to v} \otimes m_{v \to u} \tag{5.9}$$
2. *The belief generated are pairwise consistent*
3. *The normalizing constant is unique. Namely,*
$$K_u = K_v \tag{5.10}$$

Proof. We will prove 1, and from 1 we prove 2 and from 2 we prove 3. For readability we use product $\prod$ for the combination operator $\bigotimes$ and summation $\sum$ for the marginalization $\Downarrow$.
1. We have by definition that

$$m_{u \to v} = \sum_{\chi(u) - sep(u,v)} \psi_u \prod_{r \in ne(u), r \neq v} m_{r \to u}$$

Multiplying both sides by $m_{v \to u}$ we get

$$m_{u \to v}(x_{uv}) \cdot m_{v \to u}(x_{vu}) = \sum_{\chi(u) - sep(u,v)} \psi(x_u) \prod_{r \in ne(u)} m_{r \to u}(x_{ru}) = = \sum_{\chi(u) - sep(u,v)} b_u = b_{sep(u,v)} \tag{5.11}$$

which yields symmetry.
2. The property pwc follows immediately from symmetry (show as an exercise.)
3. (proof of normal constant.) If $< T, \chi, \Psi >$ is pwc relative to messages generated by CTE then

$$K_u = \sum_{\chi(u)} b_u = \sum_{sep(u,v)} \sum_{\chi(u) - sep(uv)} b_u = \sum_{sep(u,v)} b_{uv}$$

and because of pairwise consistency $b_{uv} = b_{vu}$ and therefore

$$= \sum_{sep(u,v)} b_{uv} = \sum_{sep(u,v)} b_{vu} = \sum_{sep(u,v)} \sum_{\chi(v) - sep(u,v)} b_v = \sum_{\chi(v)} b_v = K_v$$

□

If our graphical model and query are of a sum-product type, the normalizing constant is the probability of evidence or the partition function. And, as expected, we can compute this in any node in the tree decomposition. If it is the max-product or min-sum model, the normalizing constant is the cost of an optimal solution, and it also can be derived in any node.

5.3.2 COMPLEXITY OF CTE

Algorithm CTE can be subtly varied to influence its time and space complexities. The description in Figure 5.10 seems to imply an implementation whose time and space complexities are the same. Namely, that the space complexity must also be exponential in the induced-width or treewidth, denoted w. Indeed, if we compute the message in the equation in Fig. 5.10 in a brute-force manner, recording the *combined function* first, and subsequently marginalizing over the separator, we will have space complexity exponential in w. However, we can, instead, interleave the combination and marginalization operations, and thereby make the space complexity identical to the size of the sent message only, as in Fig. 5.14

```
GENERATE MESSAGES
**Input:** cluster $u$, its $\chi(u)$ and $\psi(u)$, its neighbor $v$ with $\chi(v)$, $sep = \chi(u) \cap \chi(v)$.
**Output:** the message from $u$ to $v$, $m_{u\to v}(\mathbf{x}_{sep})$.
1.  initialize: for all $\mathbf{x}_{sep}$, $m_{u\to v}(\mathbf{x}_{sep}) \leftarrow 0$.
2.      for every assignment $\mathbf{x}_{\chi(u)}$, do
3.          $m_{u\to v}(\mathbf{x}_{sep}) \Leftarrow m_{u\to v}(\mathbf{x}_{sep}) + \psi_u(\mathbf{x}_{\chi(u)}) \cdot \prod_{\{j|(j,u)\in T, j\neq v\}} m_{j\to u}(\mathbf{x}_{sep})$
4.      end for
5. Return: messages $m_{u\to v}$,
```

Figure 5.14: Generate messages.

In words, for each assignment $\mathbf{x}$ to the variables in $\chi(u)$, we compute the product functional value, and accumulate the sum value on the separator, sep, updating the message function $m_{u\to v}(\mathbf{x}_{sep})$. With this modification we now can state (and then prove) the general complexity of CTE.

Theorem 5.29 Complexity of CTE. *Given a graphical model $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \bigotimes, \Downarrow \rangle$ and its* tree-decomposition $< T, \chi, \psi >$, *where $T = (V, E)$, if N is the number of vertices in V, w its treewidth, sep its maximum separator, r the number of functions in F, deg the maximum degree in T, and k the maximum domain size of a variable, the time complexity of CTE is $O((r+N) \cdot deg \cdot k^{w+1})$ and its space complexity is $O(N \cdot k^{|sep|})$ (for a proof see appendix.)*

Trading space for time in CTE. As we noted earlier, given any tree decomposition we can generate new tree decompositions by merging adjacent clusters. While time complexity will increase, this process can generate smaller separators, and therefore CTE will require less memory.

Example 5.30 Consider the tree decompositions in Figure 5.9. For the first two decompositions CTE will have time exponential in 3 and space complexity exponential in 2. The third yields time exponential in 5 but space exponential in 1.

5.4 ILLUSTRATION OF CTE FOR SPECIFIC MODELS

In this last section of this chapter we will provide more details on algorithms tailored to specific graphical models such as Bayesian networks and constraint networks.

5.4.1 BELIEF UPDATING AND PROBABILITY OF EVIDENCE

As we saw, applying algorithm CTE to Bayesian networks when combination is *product* and the marginalization operators is *summation*, yields an algorithm that computes the explicit clusters for a given tree decomposition. In this case the explicit functions are (un-normalized) posterior marginal probability distribution given the evidence over the cluster's variables. Therefore, when the algorithm terminates the marginals can be obtained by the normalized product over all functions in the corresponding clusters. From these clusters one can also compute the probability of evidence or the posterior beliefs over singleton variables. We will refer to this specialized algorithm as *CTE-bel*. When a cluster sends a message to a neighbor, the message may contains a single *combined* function and may also contain *individual* functions that do not share variables with the relevant eliminator.

Example 5.31 Figure 5.15 describes a belief network (a) and a join-tree decomposition for it (b). Figure 5.15c shows the trace of running *CTE-bel*. In this case no individual functions appear between any of the clusters. Figure 5.15d shows the explicit output tree-decomposition. If we want to compute the probability of evidence $P(G = g_e)$, we can pick cluster 4, for example, and compute $P(G = g_e) = \sum_{e,f,g=g_e} P(g|e, f) \cdot m_{3\to4}(e, f)$ and if we wish to compute the belief for variable B for example we can use the second or the first bucket, $P(B|g_e) = \alpha \cdot \sum_{a,c} P(a) \cdot p(b|a) \cdot p(c|a, b) \cdot m_{2\to1}(b, c)$ where α is the normalizing constant that is the multiplicative inverse of the probability of evidence.

Pearl's Belief Propagation over Polytrees

A special acyclic graphical models are *polytrees*. This case deserves attention for historical reasons; it was recognized by Pearl [Pearl, 1988] as a generalization of trees on which his known belief

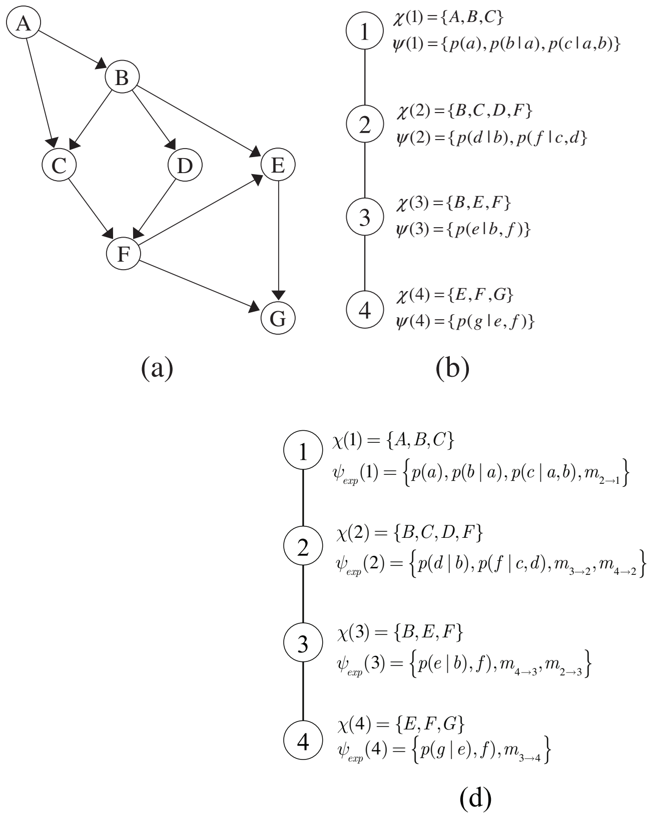

A
B
C
D
E
F
G
(a)
1
$\chi(1)=\{A,B,C\}$
$\psi(1)=\{p(a),p(b|a),p(c|a,b)\}$
2
$\chi(2)=\{B,C,D,F\}$
$\psi(2)=\{p(d|b),p(f|c,d)\}$
3
$\chi(3)=\{B,E,F\}$
$\psi(3)=\{p(e|b,f)\}$
4
$\chi(4)=\{E,F,G\}$
$\psi(4)=\{p(g|e,f)\}$
(b)
$\chi(1)=\{A,B,C\}$
$\psi_{exp}(1)=\{p(a),p(b|a),p(c|a,b),m_{2\to1}\}$
$\chi(2)=\{B,C,D,F\}$
$\psi_{exp}(2)=\{p(d|b),p(f|c,d),m_{3\to2},m_{4\to2}\}$
$\chi(3)=\{B,E,F\}$
$\psi_{exp}(3)=\{p(e|b),f),m_{4\to3},m_{2\to3}\}$
$\chi(4)=\{E,F,G\}$
$\psi_{exp}(4)=\{p(g|e),f),m_{3\to4}\}$
(d)

Figure 5.15: [Execution of CTE-bel]: (a) a belief network; (b) A join-tree decomposition; (c) execution of CTE-bel; and (d) the explicit tree-decomposition.

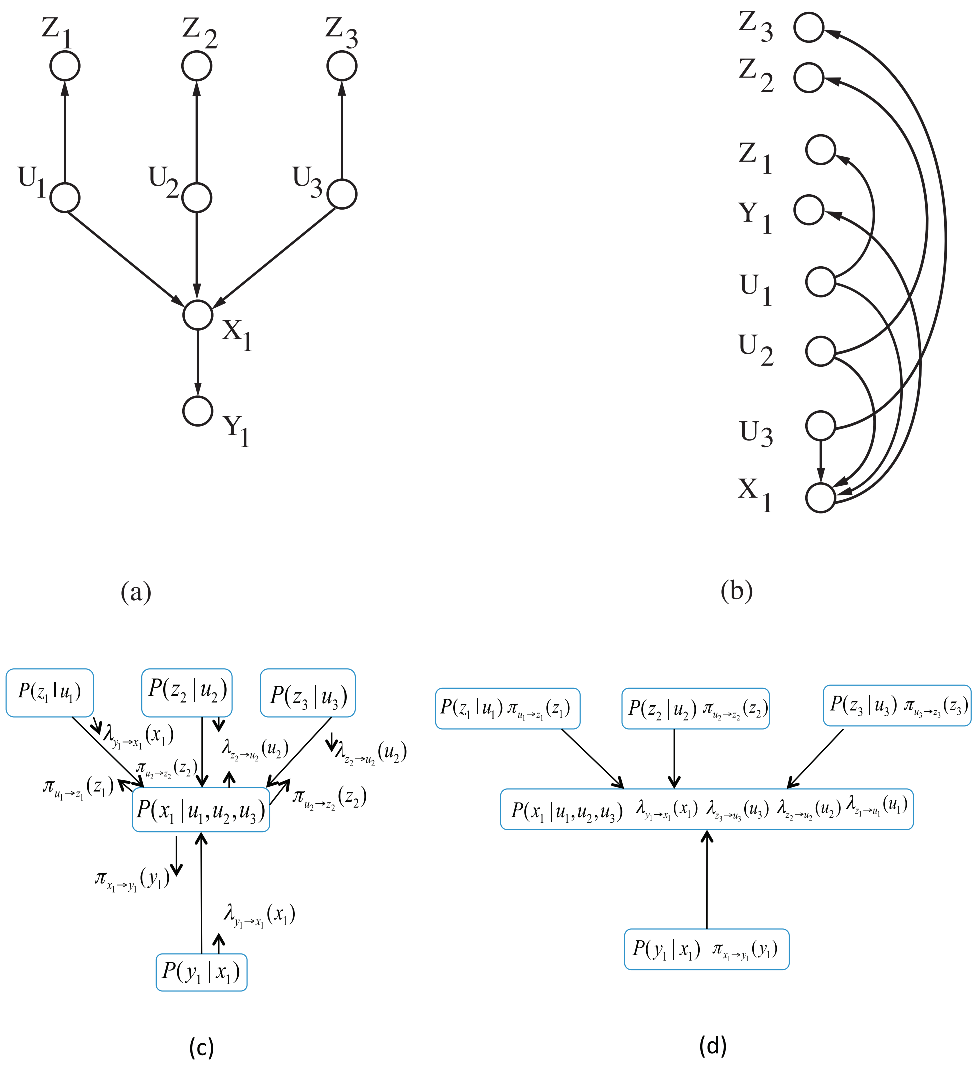
propagation algorithm was presented. It also gives rise to an iterative approximation algorithm over general networks, known as *Iterative BP* or *loopy BP* [Weiss and Pearl, 2010].

Definition 5.32 Polytree. A polytree is a directed acyclic graph whose underlying undirected graph has no cycles (see Figure 5.16a).

It is easy to see that the dual graph of a polytree is a tree, and thus yields an acyclic problem that has a join-tree decomposition where each family resides in a single node u in the decomposition. Namely $\chi(u) = \{X\} \cup pa(X)$, and $\psi(u) = \{P(X|pa(X))\}$. Note, that the separators in this *polytree decomposition*, are all singleton variables. In summary,

Proposition 5.33 *A polytree has a dual graph which is a tree and it is therefore an acyclic graphical model (Prove as an exercise.)*

It can be shown that Pearl's BP is identical to CTE if applied to the poly-tree based dual tree that is rooted in accordance with the poly-tree's topological order and where in one direction the CTE messages are named λs and in the reverse direction they are named π's.



Z1
Z2
Z3
U1
U2
U3
X1
Y1
(a)
Z3
Z2
Z1
Y1
U1
U2
U3
X1
(b)
$P(z_1|u_1)$
$P(z_2|u_2)$
$P(z_3|u_3)$
$\lambda_{y_1\to x_1}(x_1)$
$\pi_{u_2\to z_2}(z_2)$
$\lambda_{z_2\to u_2}(u_2)$
$\lambda_{z_2\to u_2}(u_2)$
$\pi_{u_1\to z_1}(z_1)$
$\pi_{u_2\to z_2}(z_2)$
$P(x_1|u_1,u_2,u_3)$
$\pi_{x_1\to y_1}(y_1)$
$\lambda_{y_1\to x_1}(x_1)$
$P(y_1|x_1)$
(c)
$P(z_1|u_1)\ \pi_{u_1\to z_1}(z_1)$
$P(z_2|u_2)\ \pi_{u_2\to z_2}(z_2)$
$P(z_3|u_3)\ \pi_{u_3\to z_3}(z_3)$
$P(x_1|u_1,u_2,u_3)\ \lambda_{y_1\to x_1}(x_1)\ \lambda_{z_3\to u_3}(u_3)\ \lambda_{z_2\to u_2}(u_2)\ \lambda_{z_1\to u_1}(u_1)$
$P(y_1|x_1)\ \pi_{x_1\to y_1}(y_1)$
(d)

Figure 5.16: (a) A polytree and (b) a legal processing ordering, (c) a polytree decomposition and messages, and (d) the explicit decomposition.

```
CLUSTER TREE-ELIMINATION (CTE-CONS)
**Input:** A tree decomposition $< T, \chi, \psi >$ for a constraint network $\mathcal{R} =< X, D, C, \bowtie, \pi >$.
$\psi_u$ is the join of original relations in cluster $u$
**Output:** An explicit tree-decomposition where each cluster is minimal (i.e., explicit).
**Compute messages:**
**for** every edge $(u, v)$ in the tree, do
    • Let $m_{u \to v}$ denote the message sent from $u$ to $v$.
          $m_{u \to v} \leftarrow \pi_{sep(u,v)}(\psi_u \bowtie (\bowtie_{R_i \in cluster_v(u)})R_i)$      (5.12)
**endfor**
**Return:** A tree-decomposition augmented with constraint messages. For every node $u \in T$, return
the decomposable (explicit) subproblem $\psi^{'}(u) = \psi(u) \cup \{m_{(i \to u)} | (i, u) \in T\}$.
If $\psi^{'}$ is consistent for every node the problem is consistent.
Generate a solution in a backtrack-free manner.
```

Figure 5.17: Algorithm cluster-tree elimination (CTE).

Example 5.34 Consider the polytree given in Figure 5.16a. An ordering along which we can run CTE is given in Figure 5.16b, a directed polytree decomposition is given in Figure 5.16c along with the π and λ messages. The explicit output tree-decomposition is given in part 5.16d. Once the propagation terminates, beliefs can be computed in each cluster.

5.4.2 CONSTRAINT NETWORKS

Algorithm CTE for constraint networks can be obtained straightforwardly by using the join operation for combination and the relational project for marginalization. The explicit algorithm called *CTE-cons* is given in Figure 5.17 (see also 9.10 in [Dechter, 2003]). It yields an *explicit* representation of the constraints in each node. This makes it possible to answer most relevant queries locally, by consulting the constraints inside each of these nodes only. This property of explicitness is called *minimality* and *decomposability* in [Montanari, 1974]as defined next.

Definition 5.35 Minimal subproblem, a decomposable network. Given a constraint problem $\mathcal{R} = (X, D, C)$, where $C = \{R_{\mathbf{S}_1}, ..., R_{\mathbf{S}_m}\}$ and a subset of variables $Y \subseteq X$, a subproblem over Y, $\mathcal{R}_Y = (Y, D_Y, C_Y)$ is minimal relative to $\mathcal{R}$, iff $sol(R_{\mathbf{Y}}) = \pi_Y sol(\mathcal{R})$ where $sol(\mathcal{R}) = \bowtie_{\mathbf{R} \in \mathcal{R}}$ $\mathbf{R}$ is the set of all solutions of network $\mathcal{R}$. A network of constraints is decomposable if each of its subnetworks is minimal.

An immediate illustration can be generated from the example in Figure 5.11. All we need is to assume that the functions are relations and the product is replaced by a *join* $\bowtie$ operator while the sum operator is replaced with the relational projection π. From the correctness of CTE it follows that:

Theorem 5.36 *Given a tree decomposition $< T, \chi, \psi >$ for a constraint network $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{C} = \{R_{\mathbf{S}_1}, ..., R_{\mathbf{S}_r}\}, \bowtie \rangle$ where $R_{\mathbf{S}_i}$ is the relation over a scope S_i. Then at termination of* CTE-cons*, the constraints in each node constitute a minimal network. Namely, for each node u, $\bowtie_{R_i \in cluster(u)} R_i = \pi_{\chi(u)}(\bowtie_{R_i \in \mathcal{R}} R_i)$.*

The minimality property is powerful. Once we have a minimal subnetwork which is not empty we immediately know that the network is consistent. Moreover, the resulting tree-decomposition whose clusters are minimal can be shown to be backtrack-free along orderings consistent with the tree structure. This means that we can generate a solution in any order along the tree and we are guaranteed to not have any dead-ends. Therefore solution generation is linear in the output network. (Exercise: prove that *CTE-cons* generates a backtrack-free ordering along some variable orderings.) Interestingly, in the case of constraints we do not need to be careful regarding the exclusion of the message sent from u to v when v computes the message it sends to u.

Counting. We have looked at algorithms for solving the tasks of computing the probability of evidence or marginals in probabilistic networks and of answering constraint satisfaction tasks over constraint networks. The first is often referred to as the *sum-product* algorithm and the second as the *join-project*. The join-project algorithm can be accomplished using Boolean operators. If we express the relations as $\{0,1\}$ cost functions(“0" for inconsistent tuple and “1" for a consistent tuple), combining functions by a Boolean product operators and marginalizing using Boolean summation will yield an identical algorithm to *CTE-cons*. If we use regular summation for the marginalization operator and a regular product operator for combination over this (0,1) cost representation of relation, then the *CTE* sum-product algorithm computes the *number of solutions* of the constraint network. Moreover, it associates any partial configuration in each cluster with the number of solutions extending it. We will refer to the algorithm as *CTE-count*.

Definition 5.37 Counts of partial solutions. Given a $\mathcal{R} =< X, D, C >$, and given a partial value assignment $\mathbf{x_S}$ over scope $\mathbf{S}$, $\mathbf{S} \subseteq \mathbf{X}$, we denote by $count(\mathbf{x_S})$, the number of full solutions of $\mathcal{R}$ that extend the partial assignment $\mathbf{x_S}$. Namely,

$$count(\mathbf{x_S}) = |\{\mathbf{y} \in sol(\mathcal{R}) | \mathbf{y_S} = \mathbf{x_S}\}| \ .$$

Theorem 5.38 *Given a tree decomposition $< T, \chi, \psi >$ for a constraint network $\mathcal{R} =< X, D, C = \{C_1, ..., C_r\} >$, where each C_i is represented as a zero-one cost function. Then when* CTE-count *terminates with $< T, \chi, \psi' >$, then for each u and for each of its assignment $\mathbf{x}_{\chi(u)}$ over its scope, $\prod_{f \in \psi'(u)} f = count(\mathbf{x}_{\chi(u)})$.*

5.4.3 OPTIMIZATION

A popular type of CTE algorithm is for solving an optimization query. Namely when $\Downarrow$ is $\max$ or $\min$. If, for example, we want to solve the MPE task over Bayesian networks, we know that we can do so by the bucket elimination algorithm when in each bucket we use the max-product combination operators. Extending that into a *CTE-max* (or a *CTE-min*, when we combine functions by summation) algorithm (called often the max-product algorithm) will generate, not only the maximum cost, but also for every partial assignment of a node u, $\mathbf{x}_{\chi(u)}$, in a tree decomposition, the maximum cost (e.g., probability) of any of its completions into a full assignment. Finally, if in our cost network the combination operator is summation, we can use min-sum or max-sum CTE. In all these cases, at termination we have what is sometimes called, *the optimal cost to go* associated with *each node* u, and therefore with each partial assignment $\mathbf{x}_{\chi(u)}$ in the tree decomposition.

For marginal map queries the extension is straightforward, we only need to make sure that the tree-decomposition will be along orderings that are legitimate. Once the clusters are generated, as join-trees for example, the messages are generated in the same manner when summation variables in each cluster should be eliminated before maximization variables.

5.5 SUMMARY AND BIBLIOGRAPHICAL NOTES

Portions of this chapter are described in more details in [Kask at. al., 2005]. Join-tree clustering was introduced in constraint processing by Dechter and Pearl [Dechter and Pearl, 1989] and in probabilistic networks by Spigelhalter et al. [Lauritzen and Spiegelhalter, 1988]. Both methods are based on the characterization by relational-database researchers that acyclic-databases have an underlying tree-structure, called join-tree, that allows polynomial query processing using join-project operations and easy identification procedures [Beeri *et al.*, 1983, Maier, 1983, Tarjan and Yannakakis, 1984]. In both constraint networks and belief networks, it was observed that the complexity of compiling any knowledge-base into an acyclic one is exponential in the cluster size, which is characterized by the induced width or tree width. At the same time, variable-elimination algorithms developed in [Bertele and Brioschi, 1972, Seidel, 1981] and [Dechter and Pearl, 1987] (e.g., adaptive-consistency and bucket-elimination) were also observed to be governed by the same complexity graph-parameter. In [Dechter and Pearl, 1987, 1989] the connection between induced-width and treewidth was recognized via the work of [Arnborg, 1985] on treewidth, k-trees and partial k-trees. This was made explicit later in [Freuder, 1992]. The similarity between variable-elimination and tree-clustering from the constraint perspective was analyzed by [Dechter and Pearl, 1989]. Independently of this investigation, the treewidth parameter was undergoing intensive investigation in the theoretic-graph-community. It characterizes the best embedding of a graph or a hypergraph in a hypertree. Various connections between hypertrees, chordal graphs and k-trees were made by Arnborg et al. [Arnborg, 1985, S. A. Arnborg and Proskourowski, 1987]. They showed that finding the smallest treewidth of a graph is NP-complete, but deciding if the graph has a treewidth below a certain constant k is polynomial in k. A recent analysis shows that this task can be accomplished in $O(n \cdot f(k))$ where $f(k)$ is a very bad exponential function of k. [Bodlaender, 1997]. The style of describing a tree-decomposition is adopted from [Georg Gottlob and Scarcello, 2000] where they talk about hypertree decompositions (not used here).

5.6 APPENDIX: PROOFS

Proof of Proposition 5.8 The input function of a tree graphical model have at most scope of 2 and therefore their size is at most k^2. Since the induced-width is just 1 for trees, meaning that the largest scope size of a bucket throughout processing, is at most 2 processing a bucket is $O(k^2)$ and the size of a message is at most $O(k)$. Since we have at most n buckets we get time and space $O(nk^2)$ and $O(nk)$, respectively. □

Proof of Theorem 5.15 (sketch)
The algorithm is linear because there exists an ordering for which each function resides alone in its bucket, and for which BTE generates messages that are subsumed by the original functions' scopes. Clearly, such messages (at most n in each direction) can be generated in time and space exponentially bounded by the functions' sizes (i.e., number of tuples in the domain of each input function). A desired ordering can be generated by processing leaf nodes along the join-tree of the acyclic model, imposing a partial order. We can show (exercise) that the ordering generated facilitates messages whose scopes are subsumed by the original function scopes which implies a linear complexity. □

Proof of Theorem 5.20
Given bucket tree $T = (V, E)$ of $\mathcal{M}$, whose nodes are mapped to buckets, we need to show how the tree can be associated with mappings χ and ψ that satisfy the conditions of Definition 5.16. In other words, the tree structure T in tree decomposition $< T, \chi, \psi >$ is the bucket tree structure, where each B_i corresponds to a vertex in V. If a bucket B_i has a parent (i.e., is connected to) bucket B_j, there is an edge $(B_i, B_j) \in E$. Labeling $\chi(B_i)$ is defined to be the union of the scopes of new and old functions in B_i during processing by BTE, and labeling $\psi(B_i)$ is defined to be the set of functions in the initial partition in B_i. With these definitions, condition 1 of Definition 5.16 is satisfied because each function is placed into exactly one bucket and also because labeling $\chi(B_i)$ is the union of scopes of all functions in B_i.

Finally, we need to prove the connectedness property (condition 2). Let's assume that there is a variable X_k with respect to which the connectedness property is violated. This means that there must be (at least) two disjoint subtrees, T_1 and T_2, of T, such that each vertex in both subtrees contains X_k, and there is no edge between a vertex in T_1 and T_2. Let B_i be a vertex in T_1 such that X_i is the earliest relative to ordering d, and B_j a vertex in T_2 such that X_j is the earliest in ordering d. Since T_1 and T_2 are disjoint, it must be that $X_i \neq X_j$. However, this is impossible since this would mean that there are two buckets that eliminate variable X_k. □

Proof of Theorem 5.25

Using the four properties of combination and marginalization operators, the claim can be proved by induction on the depth of the tree. Specifically, we will show that for every node u, CTE generates the explicit representation for that node. Namely, for every node u, $\bigotimes_{f \in \bar{\psi}(u)} f = \mathcal{M}_{\chi(u)}$.

Let $< T, \chi, \psi >$, where $T = (V, E)$ be a cluster-tree decomposition for $\mathcal{M}$, and let root it in vertex $v \in V$. We can create a partial order of the vertices of T along the rooted tree. We denote by $T_u = (V_u, E_u)$ the subtree rooted at vertex u and define by $\chi(T_u)$ all the variables associated with nodes appearing in T_u, namely: $\chi(T_u) = \bigcup_{v \in V_u} \chi(v)$.

Since commutativity permits combining the functions in any order, we select an ordering of the nodes in T $d(j) \in V, j = 1, ..., |V|$ where a vertex in the rooted tree T precedes its children in the ordering, and thus the first vertex is the root of the tree is v. As usual we denote by $\psi_u = \bigotimes_{f \in \psi(u)} f$, the combination of all the input functions in node u. Because of associativity and commutativity, clearly:

$$\forall\, u \;\; \mathcal{M}_{\chi(u)} = \Downarrow_{\chi(u)} \bigotimes_{j=1}^{|V|} \psi_{d(j)}.$$

Let u be a node having the parent w in the rooted tree T, and define $elim(u) = \chi(u) - sep(u, w)$ and $elim(T_u) = \bigcup_{v \in V_u} elim(v)$. (We will show that $elim(T_u)$ is the set of variables that are eliminated by CTE in the subtree rooted at u when sending a message to parent w). Because of the connectedness property, variables in $elim(T_u)$, appear only in the subtree rooted at u. In other words, $elim(T_u) \bigcap \{X_i | X_i \in V - \chi(T_u)\} = \emptyset$. Consequently, we can marginalize ($\Downarrow$) over such variables earlier in the process of deriving $\mathcal{M}_{\chi(u)}$ (note that $\Downarrow_{Z_i}$ means marginalizing over $X - Z_i$). If $X_i \notin \chi(u)$ and if $X_i \in elim(d(k))$ for some k, then, the marginalization eliminating X_i can be applied to $\bigotimes_{j=k}^{|V|} \psi_{d(j)}$ instead of to $\bigotimes_{j=1}^{|V|} \psi_{d(j)}$. This is safe to do, because as shown above, if a variable X_i belongs to $elim(d(k))$, then it cannot be part of any $\psi_{d(j)}$, $j < k$. We can therefore derive $\mathcal{M}_{\chi(u)}$ as follows:

$$\mathcal{M}_{\chi(u)} = \Downarrow_{\chi(u)} \bigotimes_{j=1}^{|V|} \psi_{d(j)} = \tag{5.13}$$

(because of the tree structure)

$$= \Downarrow_{\chi(u)} [\bigotimes_{j=1}^{d(k-1)} \psi_{d(j)} \Downarrow_{(X - elim(d(k)))} \bigotimes_{j=k}^{|V|} \psi_{d(j)}] = \tag{5.14}$$

$$= \Downarrow_{\chi(u)} \bigotimes_{j=1}^{d(k-1)} \psi_{d(j)} \otimes F_T(d(k)) \,, \tag{5.15}$$

where for $u = d(k)$ $F_T(u) = \Downarrow_{(X - elim(u))} \bigotimes_{j=k}^{|V|} \psi_{d(j)}$. (note that $(X - elim(u) = sep(u, w))$. We now assert that due to properties 1-4, $F_T(u)$ obeys a recursive definition relative to the tree-decomposition and that this recursive definition is identical to the messages computed by CTE and sent from u to its parent w. This is articulated by the following proposition and will conclude the proof.

Proposition 5.39 *The functions $F_T(u)$ defined above relative to the rooted tree-decomposition T, obey the following recursive definition. Let $ch(u)$ be the set of children of u in the rooted tree T.*

- *If $ch(u) = \emptyset$ (vertex u is a leaf vertex), then $F_T(u) = \Downarrow_{(X-elim(u))} \psi_u$.*
- *Otherwise, $F_T(u) = \Downarrow_{(X-elim(u))} \psi_u \otimes \bigotimes_{w \in ch(u)} F_T(w)$.*

It is easy to see that the messages computed by CTE up the tree decomposition along T are the $F_T(u)$ functions. Namely, For every node u and its parent w, $F_T(u) = m_{u \rightarrow w}$, and in particular at the root node v $F_T(v) = \mathcal{M}_{\chi(v)}$ which is identical to the message v can send to its (empty parent).

This completes the proof for the root node v. Since the argument can be applied to any node that can be made into a root of the tree, we have explicitness for all the nodes in the tree-decomposition. □

Proof of Theorem 5.29

The time complexity of processing a vertex u in tree T is $deg_u \cdot (|\psi(u)| + deg_u - 1) \cdot k^{|\chi(u)|}$, where deg_u is the degree of u, because vertex u has to send out deg_u messages, each being a combination of $(|\psi(u)| + deg_u - 1)$ functions, and requiring the enumeration of $k^{|\chi(u)|}$ combinations of values. The time complexity of CTE is

$$Time(CTE) = \sum_u deg_u \cdot (|\psi(u)| + deg_u - 1) \cdot k^{|\chi(u)|} .$$

By bounding the first occurrence of deg_u by deg and $|\chi(u)|$ by $w + 1$, we get

$$Time(CTE) \leq deg \cdot k^{w+1} \cdot \sum_u (|\psi(u)| + deg_u - 1) .$$

Since $\sum_u |\psi(u)| = r$ we can write

$$Time(CTE) \leq deg \cdot k^{w+1} \cdot (r + N)$$
$$= O((r + N) \cdot deg \cdot k^{w+1}) .$$

For each edge CTE will record two functions. Since the number of edges is bounded by N and the size of each function we record is bounded by $k^{|sep|}$.

If the cluster tree is minimal (for any u and v, $sep(u, v) \subset \chi(u)$ and $sep(u, v) \subset \chi(v)$), then we can bound the number of vertices N by n. Assuming $r \geq n$, the time complexity of CTE applied to a minimal tree-decomposition is $O(deg \cdot r \cdot k^{w+1})$. □